Process Lifecycle

- int main(void) { /* .. */ }
- int main(int argc, char* argv[]) { /* ... */ }
- int main(int argc, char* argv[], char* envp[]) { /* ... */ }

- When exec func is called, kernel needs to start the given program
- Special startup routine is called by kernel which sets up main
- C startup routine sets up environment and args into proper registers
- main returns an int which is passed to exit(3)
- When a program is started, the call could be exit(main(argc, argv))

Termination

Normal termination:

- return from mainexit(3), _exit(2), or _Exit(2)
- return of last thread from start routine
- calling pthread_exit(3) from last thread

Abnormal termination:

- calling abort(3)
- termination by signal
- response of last thread to a cancellation request

Environment

Env vars are stored in global NULL terminated array of pointers: extern char **environ; A similar array can be passed to main: int main(int argc, char **argv, char **envp)

```
#include <stdlib.h>
char *getenv(const char *name);
     Returns: value if found, NULL if not found

int putenv(char *string);
int setenv(const char *name, const char *value,
     int overwrite);
int unsetenv(const char *name);
     Returns: 0 if OK, −1 on error
```

wait(2) and waitpid(2)

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

#include <sys/resource.h>
pid_t wait3(int *status, int options, struct rusage *rusage);
pid_t wait4(pid_t pid, int *status, int options,
     struct rusage *rusage);
     Returns: child PID if OK, −1 on error
```

- wait() suspends execution of the process until status info is available for a terminated child
- waitpid() and wait(4) allow waiting for a specific process
- wait3() and wait4() allow inspection of resource usage

exec(3)

The exec() family of functions replaces the current process image with a new process image. They are all front-ends to the execve(2) system call.

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execlpe(const char *file, const char *arg, ...,
     char *const envp[]);
int execle(const char *path, const char *arg, ...,
     char *const envp[]);
int execv(const char *path, char *const argv[]);
int execve(const char *path, char *const argv[],
     char *const envp[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
     char *const envp[]);
     Returns: −1 on error, no return on success
```

Function	*pathname*	*filename*	*fd*	Arg list	*argv[]*	**environ**	*envp[]*
execl	•			•		•	
execlp		•		•		•	
execle	•			•			•
execv	•				•	•	
execvp		•			•	•	
execve	•				•		•
fexecve			•		•		•

Exits

exit(3)

```
#include <stdlib.h>
void exit(int status);
```

- terminates a process, before termination:
  - Calls the functions registered with atexit(3) in reverse order
  - Flush all open output streams, close al open streams
  - Unlink all files created with tmpfile(3) function
- calls _exit(2)

_exit(2)

```
#include <unistd.h>
void _exit(int status);
```

- terminates a process immediately
- does not call functions registered with atexit(3)

Process Control

Process ID is a nonnegative int. IDs are guaranteed to be unique and identify a particlar exisintg process.

```
#include <unistd.h>
pid_t getpid(void);   // return PID
pid_t getppid(void);  // return parent PID
uid_t getuid(void);   // return real user ID
uid_t geteuid(void);  // return effective user ID
gid_t getgid(void);   // return real group ID
gid_t getegid(void);  // return effective group ID
```

PID	Process
0	swapper (scheduler)
1	init (/sbin/init)
2	pagedaemon (virtual memory paging)
3,4,...	Other processes

fork(2)

```
#include <unistd.h>
pid_t fork(void);
     Returns: 0 in child, PID of child in parent, −1 on error
```

- fork(2) is the ONLY way to create a process in Unix kernel by user
- Child process has unique PID
- copy-on-write (COW):
  - Memory regions are read-only and shared by parent and child
  - when write occurs, kernel makes a copy of that memory for that process
- No guarantee of order of execution

Possibe reasons for fail:

- too many processes
- total # exceeds user limit

Two common uses:

- Duplicate to execute different code sections (networking)
- Execute different programs (shells)

- Parent and child share same file descriptors
- As if dup() had been called on all file descriptors
- Parent and child share same file offset
- Intermixed output from parent and child

- All processes not explicitly isntantiated by kernel were created by fork(2)
- fork(2) creates copy of current process including file descriptors and output buffers
- To replace current process with new process image, use exec(3) family of function
- After creating new process via fork(2), the parent process can wait(2) for child to process to reap its exist status and resource utilization
- Failure to wait(2) will create a zombie process until parent is terminated