Parent and Child

If the call to **fork()** is executed successfully:

- two identical copies of address space created
- one for parent, one for child
- both programs execute independently

Exit Functions

Termination Status:

- normal: exit status
- abnormal: kernel indicates reason

Child returns termination status to parent
(using wait or waitpid)

Termination Conditions

A terminated process whose parent has not
waited for is a *zombie*.

- parent can't check child's status
- Kernel keeps minimal info of child process
  (pid, status, CPU time)

If parent terminates before child:

- Kernel assigns init (pid=1) to be parent of child
- init's inherited child do not become zombies
  (wait() fetches status)

Zombies

- A zombie does not use a lot of memory
- The problem occurs when you have a lot of zombies
- Limited PIDs

System Function

Implemented by calling fork, exec, and waitpid.

```
#include <stdlib.h>
int system(const char *cmdstring);
  Returns:
    −1 with errno if fork or waitpid fails
    127 as if shell exit(127) if
      shell cannot execute command
    Termination status otherwise
```

Interpreter Files (Shebang)

Script files that begin with:

- `#!pathname [optional-argument]`
- Eg: `#!/bin/bash`, `#!/bin/csh`

Allows users an easy and efficient way to execute some commands using scripts. Ensure file is executable: `chmod +x filename`
Execing a script file: `execl("/bin/testinterp", "testinterp", "myarg1", NULL);`

Threads

One process can have multithreads

- Each thread handles a separate task
- Threads have access to same memory address and file descriptors
- Multithreaded process can run on a uniprocessor
- Ex for word processor:
  - Background thread checks spelling / grammar
  - Foreground thread handles user input
  - Third thread loads images from hard drive
  - Fourth thread does automatic saves

Identification

A thread's ID is represented by `pthread_t` type. The `pthread_equal` function is used to compare two IDs.

```
#include <pthread.h>
int pthread_equal(pthread_t tid1, pthread_t tid2);
    Returns:
      nonzero if equal, 0 otherwise
pthread_t pthread_self(void);
    Returns
      thread ID of calling thread
```

Creation (pthread_create(3))

```
#include <pthread.h>
int pthread_create(pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_rtn)(void *), void *arg);
```

- `tidp`: new thread id
- `attr`: thread attributes
- `start_rtn`: function to be executed
- `arg`: argument to be passed to `start_rtn`

Termination

If any thread in a process calls `exit`, `_exit`, or `_Exit`, the entire process terminates. When default action is to terminate the process, a `signal` sent to a thread will terminate the entire process. A single thread can exit in three ways without affecting the entire process:

1. Thread can `return` from start routine, returned value is thread's exit status
2. Thread can be `canceled` by another thread in same process
3. Thread can call `pthread_exit`

```
#include <pthread.h>
void pthread_exit(void *rval_ptr);
int pthread_join(pthread_t thread, void **rval_ptr);
    /* similar to wait */
    Returns: 0 if successful, error number otherwise
int pthread_cancel(pthread_t thread);
    /* like pthread_exit with arg of PTHREAD_CANCELED */
    Returns: 0 if successful, error number otherwise
```

Process primitive	Thread primitive	Description
`fork`	`pthread_create`	Create new flow of control
`exit`	`pthread_exit`	Exit from existing flow of control
`waitpid`	`pthread_join`	Wait for flow of control to terminate
`atexit`	`pthread_cleanup_push`	Register function to be called at exit
`getpid`	`pthread_self`	Get ID of flow of control
`abort`	`pthread_cancel`	Terminate flow of control

Cleanup

```
#include <pthread.h>
void pthread_cleanup_push(void (*rtn)(void *), void *arg);
  /* Called when thread exits */
void pthread_cleanup_pop(int execute);
  /* Removes cleanup handler establish by last
     call to pthread_cleanup_push */
int pthread_detach(pthread_t thread);
```