```c
#include <iostream.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <main.h>

int main(void)
{
    int pid;
    int theInt=42;
    pid = fork();

    // Error occurred
    if (pid < 0) {
        cerr << "main: Fork failed!" << endl;
        exit(-1);
    }

    // child process
    else if (pid == 0) {
        cout << "theInt is: " << theInt << endl;
        execlp("/bin/ls", 'ls', NULL);
    }

    // parent process wait for child to terminate
    else {
        wait(NULL);
        cout << "Child complete" << endl;
        exit(0);
    }

    return 1;
}
```
```cpp
#include <iostream.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>

int main(void)
{
  int pid;
  int theInt=42;

  pid = fork();

  // Error occurred
  if (pid < 0) {
    cerr << "main: Fork failed!" << endl;
    exit(-1);
  }

  // Child process
  else if (pid == 0) {
    cout << "theInt is: " << theInt << endl;
    sleep(20);
  }

  // Parent process wait for child to terminate
  else {
    cout << "Exiting early..." << endl;
    exit(0);
  }

  return 1;
}
```
Processes Tree on a UNIX System
### Sample Process Control Block

<table>
<thead>
<tr>
<th>Process management</th>
<th>Memory management</th>
<th>File management</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>Pointer to text segment</td>
<td>Root directory</td>
</tr>
<tr>
<td>Program counter</td>
<td>Pointer to data segment</td>
<td>Working directory</td>
</tr>
<tr>
<td>Program status word</td>
<td>Pointer to stack segment</td>
<td>File descriptors</td>
</tr>
<tr>
<td>Stack pointer</td>
<td></td>
<td>User ID</td>
</tr>
<tr>
<td>Process state</td>
<td></td>
<td>Group ID</td>
</tr>
<tr>
<td>Priority</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Scheduling parameters</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process ID</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Parent process</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Process group</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Signals</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time when process started</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU time used</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Children’s CPU time</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time of next alarm</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2-4.** Some of the fields of a typical process table entry.
“get_info” is called at “read”, while “owner” is used to protect modules from unlocking while proc_dir_entry is in use.

```c
typedef int (read_proc_t)(char *page, char **start, off_t off,
                         int count, int *ref, void *data);
typedef int (write_proc_t)(struct file *file, const char *buffer,
                         unsigned long count, void *data);
typedef int (get_info_t)(char *, char **, off_t, int);

struct proc_dir_entry {
    unsigned short low_ino;
    unsigned short namelen;
    const char *name;
    mode_t mode;
    nlink_t nlink;
    uid_t uid;
    gid_t gid;
    unsigned long size;
    struct ino_operations * proc_iops;
    struct file_operations * proc_fops;
    get_info_t *get_info;
    struct module *owner;
    struct proc_dir_entry *next, *parent, *subdir;
    void *data;
    read_proc_t *read_proc;
    write_proc_t *write_proc;
    atomic_t count; /* use count */
    int deleted;  /* delete flag */
    kdev_t cdev;
};
```

#define PROC_INODE_PROPER(inode) ((inode)->i_ino & ~Oxffff)

#define CONFIG_PROC_FS

extern struct proc_dir_entry proc_root;
extern struct proc_dir_entry *proc_root_fs;
extern struct proc_dir_entry *proc_net;
extern struct proc_dir_entry *proc_bus;
extern struct proc_dir_entry *proc_root_driver;
extern struct proc_dir_entry *proc_root_kcore;

extern void proc_root_init(void);
```