

Process Management

CS351 : Saelee

Creation

```
#include <unistd.h>
```

```
pid_t fork(void);
```

Called once, Returns twice

Once to “parent”
Process ID of child

Once to “child”

∅

Parent == Child

(but separate)

```
int main () {  
    fork();  
    printf("Hello world!\n");  
}
```

```
Hello world!  
Hello world!
```

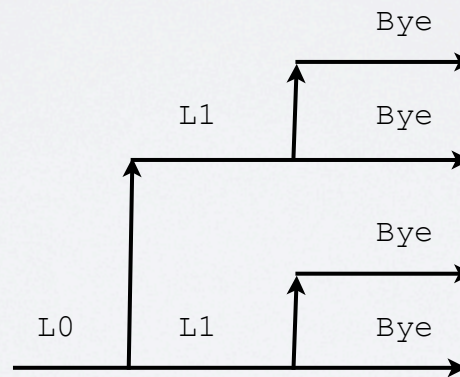
```
int main () {  
    fork();  
    fork();  
    printf("Hello world!\n");  
}
```

```
Hello world!  
Hello world!  
Hello world!  
Hello world!
```

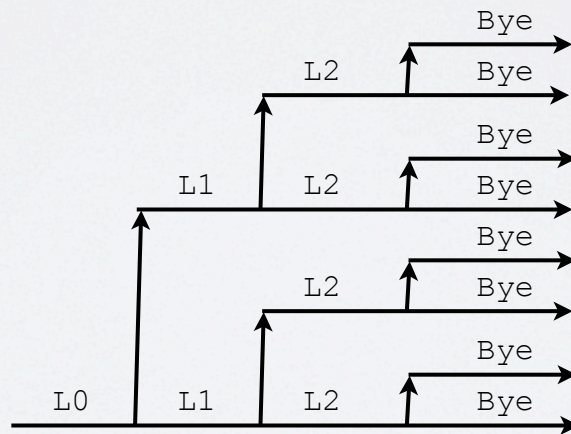
```
void fork1 () {  
    int x = 1,  
        pid = fork();  
    if (pid == 0) {  
        printf("Child has x = %d\n", ++x);  
    } else {  
        printf("Parent has x = %d\n", --x);  
    }  
}
```

```
Parent has x = 0  
Child has x = 2
```

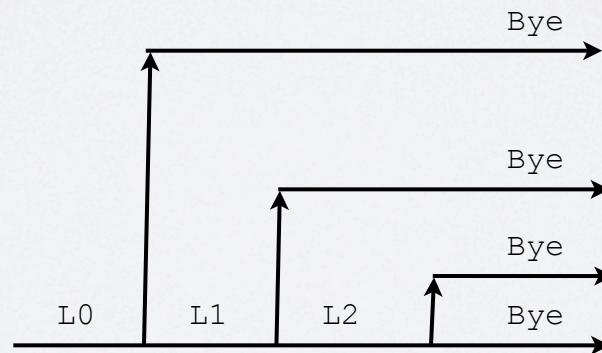
```
void fork2 () {  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("Bye\n");  
}
```



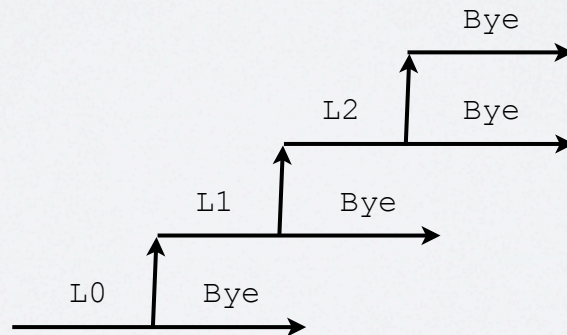
```
void fork3 () {  
    printf("L0\n");  
    fork();  
    printf("L1\n");  
    fork();  
    printf("L2\n");  
    fork();  
    printf("Bye\n");  
}
```



```
void fork4 () {  
    printf("L0\n");  
    if (fork() != 0) {  
        printf("L1\n");  
        if (fork() != 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



```
void fork5 () {  
    printf("L0\n");  
    if (fork() == 0) {  
        printf("L1\n");  
        if (fork() == 0) {  
            printf("L2\n");  
            fork();  
        }  
    }  
    printf("Bye\n");  
}
```



Destruction

```
int main () {  
    return 0;  
}
```

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

```
int atexit(void (*func)(void));
```

```
void cleanup (void) {  
    printf("cleaning up\n");  
}
```

```
void fork6 () {  
    atexit(cleanup);  
    fork();  
    exit(0);  
}
```

```
cleaning up  
cleaning up
```

Zombies

not active
still alive

consume system resources
kernel data structures

Reaping

who?


Parent reaps Child

orphaned children?

inherited by “proc 0”
reaped by kernel

```
void fork7 () {  
    if (fork() == 0) {  
        /* Child */  
        printf("Terminating Child, PID=%d\n", getpid());  
        exit(0);  
    } else {  
        printf("Running Parent, PID=%d\n", getpid());  
        while (1); /* Infinite loop */  
    }  
}
```

(demo)



Use ps -0 status

```
void fork8 () {  
    if (fork() == 0) {  
        /* Child */  
        printf("Running Child, PID=%d\n", getpid());  
        while (1); /* Infinite loop */  
    } else {  
        printf("Terminating Parent, PID=%d\n", getpid());  
        exit(0);  
    }  
}
```

(demo)

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

1. wait until child terminates
2. reap
3. get child status (optional)

```
void fork9() {  
    int child_status;  
  
    if (fork() == 0) {  
        printf("HC: hello from child\n");  
    }  
    else {  
        printf("HP: hello from parent\n");  
        wait(&child_status);  
        printf("CT: child has terminated\n");  
    }  
    printf("Bye\n");  
  
    exit();  
}
```



synchronization

```
int main() {
    if (fork() == 0) {
        if (fork() == 0) {
            printf("3");
        } else {
            int status;
            wait(&status);
            printf("4");
        }
    } else {
        if (fork() == 0) {
            printf("1");
            exit(0);
        }
        printf("2");
    }
    printf("0");
    return 0;
}
```

Possible outputs?

A. 2030401

B. 1234000

C. 2300140

D. 2034012

E. 3200410

F. 3401200

wait macros

WIFEXITED(status)

WEXITSTATUS(status)

WIFSIGNALED(status)

WIFSTOPPED(status)

WTERMSIG(status)

```
void fork10 () {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", wpid);
    }
}
```

```
Child 7079 terminated with exit status 100
Child 7080 terminated with exit status 101
Child 7083 terminated with exit status 104
Child 7082 terminated with exit status 103
Child 7081 terminated with exit status 102
```

explicit reaping: `waitpid`

```
pid_t waitpid(pid_t pid,  
              int *status,  
              int options);
```

```
options = {WNOHANG, WUNTRACED}
```

```
void fork11 () {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n",
                wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

```
Child 7085 terminated with exit status 100
Child 7086 terminated with exit status 101
Child 7087 terminated with exit status 102
Child 7088 terminated with exit status 103
Child 7089 terminated with exit status 104
```

Running new programs

exec family

```
#include <unistd.h>
```

```
int execl(const char *path, const char *arg, ...);  
int execlp(const char *file, const char *arg, ...);  
int execv(const char *path, char *const argv[]);  
int execvp(const char *file, char *const argv[]);
```

replace process image
keep environment

```
int main () {  
    if (fork() == 0) {  
        execl("/bin/ls", "/bin/ls", (void *) 0);  
    }  
    wait(NULL);  
    printf("Command completed\n");  
    return 0;  
}
```

```
$ ./a.out  
test.c a.out  
Command completed
```

no return on success!

Demo

Demo "exec" utility

- try exec ls
- try exec sleep 3