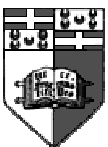




Process Identifiers

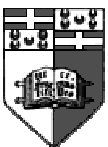
- Every process apart from the PID also has a PUID and a PGID.
- There are two types of PUID and PGID: real and effective.
- The real PUID is always equal to the user running the process and the same for PGID.
- The effective ID gives us the rights of that user.
- Also every process has a parent process which has the relevant process and owner IDs.

```
#include <sys/types.h>
#include <unistd.h>
pid_t getpid(void);
pid_t getppid(void);
uid_t getuid(void);
uid_t geteuid(void);
gid_t getgid(void);
gid_t getegid(void);
```



Unix Parents and Children

- In UNIX, the system initially creates three special processes: *init* (PID no.1), *swapper* (*scheduler*) and *pagedaemon*.
- *init* is the parent of all processes. It spawns off all other processes, including the terminal login process. It is a normal user process with **superuser** privileges.
- These **superuser** privileges are changed upon login.
- Each process is able to spawn off as many child processes as desired using *fork()*.

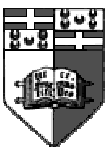




Process Creation

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
        returns 0 in child
                child PID in parent
                -1 on error
```

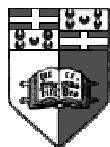
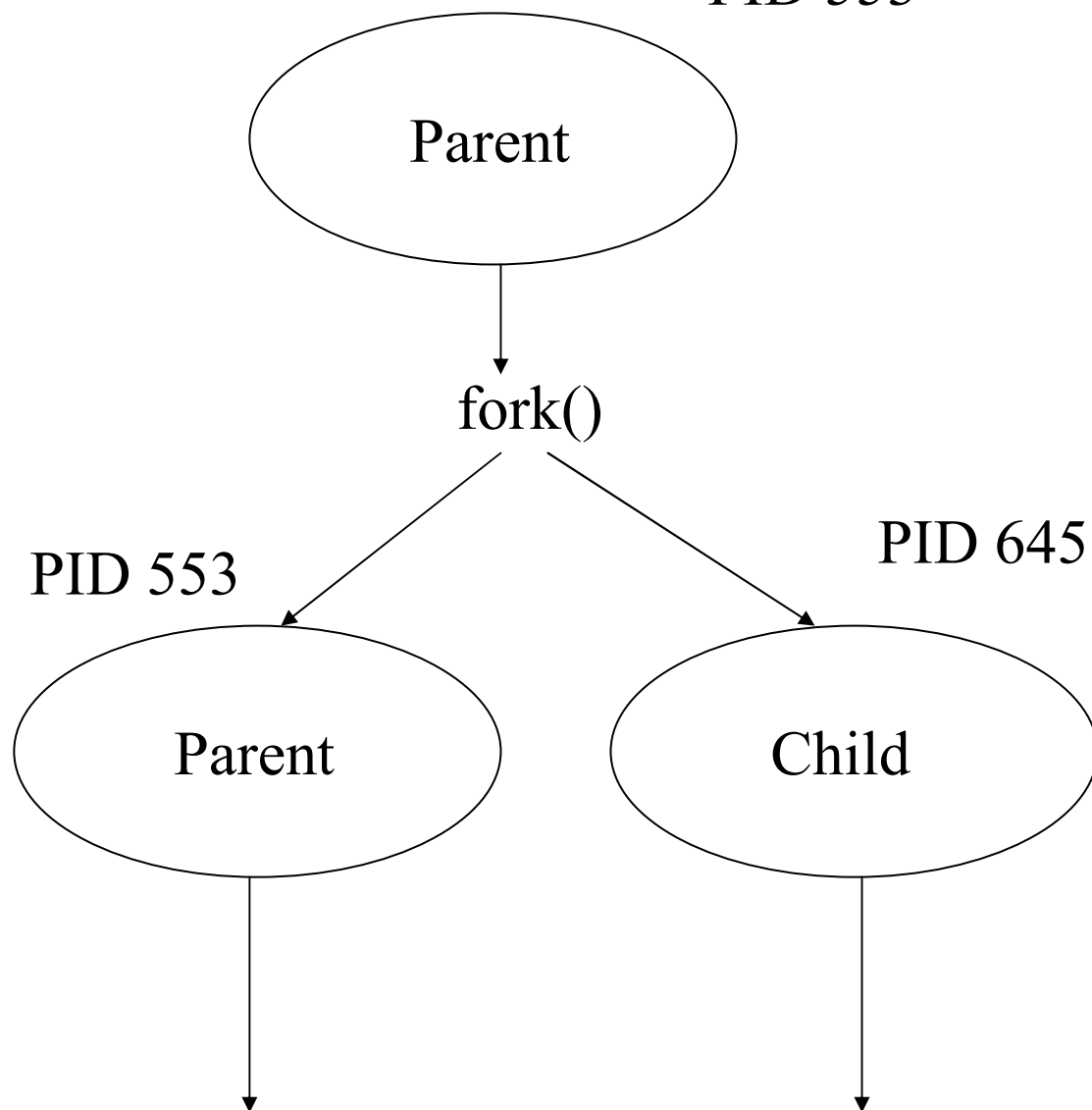
- *fork()* allocates new space for a new process, copies the parent's memory layout in this space and creates the required descriptors for this process to execute.
- The child process is assigned a new PID.
- Execution of the child process starts from inside *fork()*.
- The return value of *fork()* differs for the child and for the parent.





Process Creation

PID 553





Child Inheritance

- Child process inherits these from the parents process
 - UID, GID (real and effective)
 - Session ID
 - Pointers to same file table entries
 - current terminal
 - SUID and SGID flag
 - current and root directory
 - file mode creation mask
 - environment
 - FD_CLOEXEC flag
 - resource limits
 - signal mask
 - shared memory segments

Parent

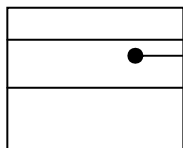
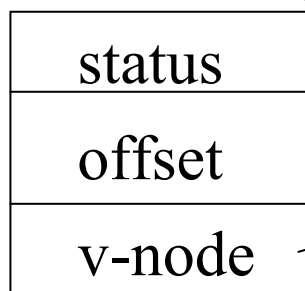
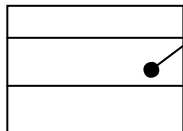
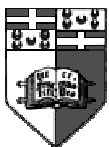


Table Entry



Child



Joseph Cordina

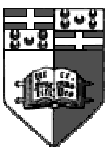


Unix Child Inheritance (cont)

- It is important to note that all files are shared between child and parent.
- This also applies to standard output, standard input and standard error.
- Yet obviously closing a file from one process will not close it for the other process.
- **Race conditions** become a serious consideration in spawned off processes.

Using *fork()*

```
main() {  
    int pid ;  
    pid = fork();  
    if (pid == 0) {  
        child stuff  
    } else {  
        parent stuff }  
}
```

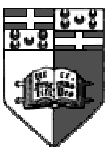




Executing

With just spawning, the new process cannot do much.

- Thus normally after every spawn, the child process loads a new program and executes it..
- The *exec()* functions load an executable into the current process memory space, overwriting the previous contents and execute it..
- After an *exec()* call the following are preserved from the calling process
 - PID and PPID
 - real UID and GID (not effective)
 - terminal
 - file creation mask
 - resource limits
 - signal mask
 - root and current working directory
 - file locks
 - Open files depending on `FD_CLOEXEC`

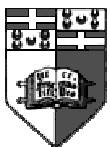




Executing (cont)

```
#include <unistd.h>
int execl(const char *pathname, const char *arg0,
          ..., 0);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg0,
          ..., 0, char *const envp[]);
int execve(const char *pathname, char *const argv[],
           char *const envp[]);
int execlp(const char *filename, const char *arg0,
           ..., 0);
int execvp(const char *filename, char *const argv[]);
           return -1 on error, no return on success!!
```

- The first four formats take a direct path while the last two search for filename in the PATH environment variable.
- Note that after an *exec()* the same PID is maintained but all the user process memory layout is overwritten.



example : `execl("/bin/ls", "-l", (char *) 0);`

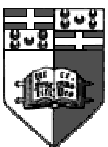




Close On Execute

- The `FD_CLOEXEC` is set for an open file using `fcntl()` and if set, the file will be automatically closed after the `exec()` call.
- Default is to leave the file open.

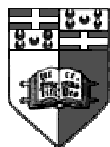
```
#include <sys/types.h>
#include <unistd.h>
#include <fcntl.h>
int fcntl(int fd, F_GETFD);
           return value of flag
int fcntl(int fd, F_SETFD, 0 or 1);
           return 0 if OK, -1 on error
```





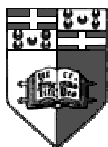
COW and *vfork()*

- Normally when a new child process is spawned using a *fork()*, it is followed by one of the *exec()* functions.
- So what is the use of making a copy of the parent's memory when it is going to be overwritten shortly?
- Thus most implementations of *fork()* use **C**opy **O**n **W**rite, where the same memory space is shared until the child or parent process try to write to this area.
- *vfork()* on the other hand creates a new child process but still uses up the same address space until a call to an *exec()* function occurs. This call is expected to occur after the *fork()*.
- Also *vfork()* guarantees that the parent will be suspended until the child calls *exec()* or *exit()*;



Unix Process Termination

- When a child process terminates it sends a signal (SIGCHLD) to the parent.
- The child remains to exist until the parent reaps this signal. Until this happens, the child is a **zombie process**.
- If the parent reaps this termination signal, it receives the termination status of the child and the child stops to exist.
- If the parent terminates before the child, the child's parent becomes the *init()* process (*init* inherits the child).
- If the child terminates with *init()* as its parent, *init()* reaps its termination signal immediately, stopping the system from being hogged with zombie processes.

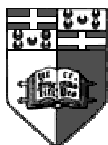




Waiting for Child Termination

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *statloc);
        return child ID if ok, -1 on error
```

- When parent calls *wait()* it blocks until a child terminates.
- If a child zombie process exists, *wait()* returns immediately.
- Exit status of child is stored inside **statloc* together with special status bits
 - `WIFEXITED(stat) ◇ WEXITSTATUS(stat)`
 - `WIFSIGNALED(stat) ◇ WTERMSIG(stat)`
 - `WIFSTOPPED(stat) ◇ WSTOPSIG(stat)`
- *wait()* has to be called for each and every child to clean up the system.

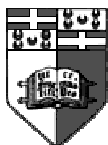




Waiting for Child Termination (cont)

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statloc, int options);
        return child ID if ok, -1 on error or 0
```

- *waitpid()* waits for a specific child to terminate if *pid* > 0.
- If *pid* is -1, then it waits for any child process, like *wait()*.
- If *pid* is 0 then it waits for any child with same PGID.
- Options can be 0 or WNOHANG.
- With WNOHANG, if no child zombie process exists, *waitpid()* returns immediately (does not block) with return value of 0.

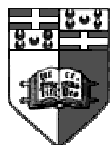




Set User ID

- The SUID and SGID is another flag in the file access permission.
- When a user *exec()*'s a process with the SUID flag set, the process seems to be owned by the file owner rather than by the caller of *exec()*.
- The system stores three UIDs and GIDs per process:
 - Effective (what is really used)
 - Real
 - Saved

	<i>exec()</i>	
	SUID OFF	SUID ON
real	unchanged	unchanged
effective	unchanged	File UID
saved	real	File UID



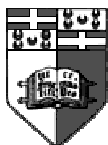
Unix Set User ID (cont)

- The SUID and SGID file flag are set using the S_ISUID and S_ISGID mode field of *chmod()*.

```
#include <sys/types.h>
#include <sys/stat.h>
int chmod(const char *pathname, mode_t mode);
int fchmod(int fd, mode_t mode);
           returns 0 if OK, -1 on error
```

- A user can change the effective UID and GID between the real and saved values.
- When the **superuser** calls *setuid()*, all three UIDs and all three GIDs are set.

```
#include <sys/types.h>
#include <unistd.h>
int setuid(uid_t uid);
int setgid(gid_t gid);
           return -1 on error, 0 when OK
```



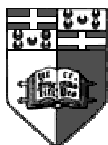


Miscellaneous Functions

```
#include <stdlib.h>
int system(const char *cmdstring);
    return value depends on operation requested
```

```
#include <unistd.h>
char *getlogin(void);
    return NULL on error
```

- *system()* allows the user to execute a command string from within the program.
- *getlogin()* returns the name under which the user logged on the system.





Exercises

- Write a process which spawns off a number of children processes keeping a record of each PID. Terminate the parent process only when all children have terminated.
- What happens to the value of a variable in the parent if a child changes its value after a *vfork()* but before an *exec()* call? Try it.
- Implement a **race condition** using *fork()*'ed children.
- Implement your own *system()* function call assuming the calls given don't contain command line arguments.

