# **int ForkExec(char \* file)**

**DESCRIPTION**

The **ForkExec**() creates a new child process running in separate memory space with *file* the file being executed.

The parent process will wait for termination of all children processes.

IDs are unique.

**RETURN VALUE**

On success, **ForkExec**() returns the child process ID.

On error, it returns a negative error number.

**ERRORS**

**-1** *file* doesn’t exist or doesn’t have execute permission

# **int ForkWait(int pid)**

**DESCRIPTION**

The **ForkWait**() function suspends execution of the calling thread until his child process identified with *pid* terminates. If that process has already terminated, then **ForkWait**() returns immediately.

Only children of the calling process can be waited. Therefore, the initial process with process ID 0 cannot be waited.

**RETURN VALUE**

On success, **ForkWait**() returns 0.

On error, it returns a negative error number.

**ERRORS**

**-1** Process *pid* doesn’t exist

# **int GetPID(void)**

# **int GetPPID(void)**

**DESCRIPTION**

**GetPID**() returns the process ID of the calling process.

**GetPPID**() returns the process ID of the parent of the calling process.

**int UserThreadCreate(void (\*f)(void \*), void \* arg)**

**DESCRIPTION**

The **UserThreadCreate**() function starts a new thread in the calling process. The new thread starts execution by invoking

*f*(); *arg* is passed as the sole argument of *f*().

The new thread terminates either if it calls **UserThreadExit()** or if it reach the end of*f*().

Main thread will wait for termination of all threads in the process.

IDs are unique per process.

**RETURN VALUE**

On success, **UserThreadCreate**() returns the thread ID.

On error, it returns a negative error number.

**ERRORS**

**-1** Insufficient stack memory

**-2** System’s limit on number of threads was reached

# **void UserThreadExit(void)**

**DESCRIPTION**

The **UserThreadExit**() function terminates the calling thread.

When a thread terminates, process-shared resources are not

Released until the main thread exit.

Implicitly called with return.

# **int UserThreadJoin(int tid)**

**DESCRIPTION**

The **UserThreadJoin**() waits for the thread specified by *tid* to terminate. If that thread has already terminated, then

**UserThreadJoin**() returns immediately.

The main thread cannot be joined.

Multiple threads can simultaneously try to join with the same thread.

**RETURN VALUE**

On success, **UserThreadJoin**() returns 0.

On error, it returns a negative error number.

**ERRORS**

**-1** Thread *tid* doesn’t exist

# **void PutChar(char c)**

# **void PutString(char \* s)**

# **void PutInt(int n)**

**DESCRIPTION**

**PutChar**() writes the character *c* to *stdout*.

**PutString**() writes the string *s*, without its terminating null byte ('\0'), to *stdout*. A maximum of **MAX\_STRING\_SIZE** characters are written; the remaining will be truncated.

**PutInt**() writes converted *n* to *stdout*.

# **char GetChar(void)**

**DESCRIPTION**

**GetChar**() reads a character from *stdin*.

**RETURN VALUE**

Upon successful completion, *fgetc*() shall return the next byte from the *stdin*.

If *stdin* is closed, returns **EOF**.

# **void GetString(char \* s, int n)**

**DESCRIPTION**

**GetString**() reads *n*-1 characters from *stdin* into the buffer pointed to by *s* until either a terminating newline or **EOF**.

A null byte ('\0') is then written.

No check for buffer overrun is performed.

# **int GetInt(void)**

**DESCRIPTION**

**GetInt**() converts characters from a **GetString()** call into a signed decimal value according to **sscanf()** %d format.

**RETURN VALUE**

The **GetInt**() function returns the result of the conversion.

If no valid conversion could be performed, 0 is returned.

# **int UserMutexLock(int id)**

**DESCRIPTION**

The mutex object referenced by *id* shall be locked.

If the mutex is already locked by another thread, the calling thread shall block.

If a thread attempts to relock a mutex that it has already locked, then **UserMutexLock**() returns immediately.

**RETURN VALUE**

On success, **UserMutexLock**() returns 0.

On error, it returns a negative error number.

**ERRORS**

**-1** The current thread already owns the mutex or didn’t exist.

# **int UserMutexUnlock(int id)**

**DESCRIPTION**

The mutex object referenced by *id* shall be unlocked.

If there are threads blocked on the mutex when **UserMutexUnlock**() is called, the scheduling policy shall determine which thread shall acquire the mutex.

If a thread attempts to unlock a mutex that it is not locked by any thread, then **UserMutexUnlock**() returns immediately.

**RETURN VALUE**

On success, **UserMutexUnlock**() returns 0.

On error, it returns a negative error number.

**ERRORS**

**-1** The mutex was already unlocked or didn’t exist.

# **int UserMutexDestroy(int id)**

**DESCRIPTION**

The *UserMutexDestroy*() function shall destroy the mutex referenced by *id*.

Attempting to destroy a locked mutex results in undefined behavior.

**RETURN VALUE**

On success, **UserMutexDestroy**() returns 0.

On error, it returns a negative error number.

**ERRORS**

**-1** The mutex didn’t exist.

# **int UserMutexCreate(void)**

**DESCRIPTION**

The *UserMutexCreate*() function shall initialize the mutex.

**RETURN VALUE**

**UserMutexCreate**() returns the mutex’s ID.

…