Embedded Systems Programming

Process and Thread

Iwona Kochańska

Katedra Systemów Elektroniki Morskiej
WETI PG
Process

- The **fork() system call** allows one **process**, the **parent**, to create a new process, the **child**.
  - new child process is (almost) exact duplicate of the parent: the child obtains copies of the parent’s stack, data, heap, and text segments

- The **exit(status) library function** terminates a process, making all resources (memory, open file descriptors, and so on) used by the process available for subsequent reallocation by the kernel.
  - The status argument is an integer that determines the termination status for the process.
  - Using the **wait()** system call, the parent can retrieve this status
  - The **exit() library function** is layered on top of the _exit() system call.
  - Generally only one of the parent and child terminate by calling exit(); the other process should terminate using _exit()
Process

- The **wait(&status) system call** has two purposes.
  - First, if a child of this process has not yet terminated by calling `exit()`, then `wait()` suspends execution of the process until one of its children has terminated.
  - Second, the termination status of the child is returned in the status argument of `wait()`.

- The **execve(pathname, argv, envp) system call** loads a new program (pathname, with argument list argv, and environment list envp) into a process’s memory.
  - The existing program text is discarded, and the stack, data, and heap segments are freshly created for the new program.
  - This operation is often referred to as **executing a new program**.
  - Interface variations are sometimes called as `exec()`, but there is no system call or library function with this name.
Process

Figure: Use of fork(), exit(), wait(), and execve() [1].
Process creation: fork()

▸ prototype:

```c
#include <unistd.h>
pid_t fork(void);
```

▸ In parent: returns process ID of child on success, or –1 on error

▸ in successfully created child: always returns 0

▸ After fork() has completed its work, two processes exist, and, in each process, execution continues from the point where fork() returns.

▸ The two processes are executing the same program text, but they have separate copies of the stack, data, and heap segments.

▸ The child’s stack, data, and heap segments are initially exact duplicates of the corresponding parts the parent’s memory.

▸ After the fork(), each process can modify the variables in its stack, data, and heap segments without affecting the other process.
Process creation: fork()

- template often used [1]:

```c
pid_t childPid;       /* Used in parent after successful fork() to record PID of child */
switch (childPid = fork()) {
  case -1:            /* fork() failed */
    /* Handle error */
  case 0:             /* Child of successful fork() comes here */
    /* Perform actions specific to child */
  default:            /* Parent comes here after successful fork() */
    /* Perform actions specific to parent */
}
```
Process creation: fork()

- Performing a simple copy (by fork()) of the parent’s virtual memory pages into the new child process would be wasteful.

- fork() is often followed by an immediate exec(), which replaces the process’s text with a new program and reinitializes the process’s data, heap, and stack segments. Most modern UNIX implementations, including Linux, use two techniques to avoid such wasteful copying:
  - The kernel marks the text segment of each process as read-only, so that a process can’t modify its own code. This means that the parent and child can share the same text segment. The fork() system call creates a text segment for the child by building a set of per-process page-table entries that refer to the same virtual memory page frames already used by the parent.
  - For the pages in the data, heap, and stack segments of the parent process, the kernel employs copy-on-write.
Process creation: fork()

**Figure:** Page tables before and after modification of a shared copy-on-write page [1]
Process creation: vfork()

- Historically, early BSD implementations were among those in which fork() performed a literal duplication of the parent’s data, heap, and stack.
- Later versions of BSD introduced the vfork() system call, which was far more efficient than BSD’s fork(), although it operated with slightly different semantics.
- Modern UNIX implementations employing copy-on-write for implementing fork() are much more efficient than older fork() implementations, thus largely eliminating the need for vfork().
- Nevertheless, Linux provides a vfork() system call with BSD semantics for programs that require the fastest possible fork.
- Because the unusual semantics of vfork() can lead to some subtle program bugs, its use should normally be avoided, except in the rare cases where it provides worthwhile performance gains.
Prototype:

```c
#include <unistd.h>
pid_t vfork(void);
```

- In parent: returns process ID of child on success, or -1 on error.
- In successfully created child: always returns 0.
- Two features distinguish the vfork() system call from fork() and make it more efficient:
  - No duplication of virtual memory pages or page tables is done for the child process. Instead, the child shares the parent's memory until it either performs a successful exec() or calls _exit() to terminate.
  - Execution of the parent process is suspended until the child has performed an exec() or _exit().
Process termination

- A process may terminate in two general ways:
  - abnormal termination, caused by the delivery of a signal whose default action is to terminate the process
  - normal termination, using the _exit() system call.
- prototype of _exit():

  ```c
  #include <unistd.h>
  void _exit(int status);
  ```

  - int status - defines the termination status of the process, which is available to the parent of this process when it calls wait().
  - although defined as an int, only the bottom 8 bits of status are made available to the parent (0-255).
  - status of 0 indicates that a process completed successfully,
  - nonzero status value indicates that the process terminated unsuccessfully.
  - There are no fixed rules about how nonzero status values are to be interpreted; different applications follow their own conventions, which should be described in their documentation.
  - A process is always successfully terminated by _exit() (i.e., _exit() never returns).
Process termination

- Programs generally don’t call `_exit()` directly, but instead call the `exit()` library function, which performs various actions before calling `_exit()`.

- Prototype of `exit()`:

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

- Actions are performed by `exit()`:
  - Exit handlers (functions registered with `atexit()` and `on_exit()`) are called, in reverse order of their registration.
  - The stdio stream buffers are flushed.
  - The `_exit()` system call is invoked, using the value supplied in `status`.

- Unlike `_exit()`, which is UNIX-specific, `exit()` is defined as part of the standard C library; that is, it is available with every C implementation.

- Performing an explicit `return n` at the end of `main()` function is equivalent to calling `exit(n)` (run-time function that invokes `main()` uses the return value from `main()` in a call to `exit()`).
In many applications where a parent creates child processes, it is useful for the parent to be able to monitor the children to find out when and how they terminate.

The **wait() system call** waits for one of the children of the calling process to terminate and returns the termination status of that child in the buffer pointed to by status.

**Prototype:**

```c
#include <sys/wait.h>

pid_t wait(int *status);
```

Returns process ID of terminated child, or –1 on error

One possible error is that the calling process has no children, which is indicated by the errno value ECHILD
Monitoring child processes: wait()

The wait() system call does the following:

1. If no (previously unwaited-for) child of the calling process has yet terminated, the call blocks until one of the children terminates. If a child has already terminated by the time of the call, wait() returns immediately.

2. If status is not NULL, information about how the child terminated is returned in the integer to which status points.

3. The kernel adds the process CPU times and resource usage statistics to running totals for all children of this parent process.

4. As its function result, wait() returns the process ID of the child that has terminated.
Monitoring child processes: waitpid()

The wait() system call has a number of limitations, which \textit{waitpid()} was designed to address:

- If a parent process has created multiple children, it is not possible to wait() for the completion of a specific child; we can only wait for the next child that terminates.

- If no child has yet terminated, wait() always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.

- Using wait(), we can find out only about children that have terminated. It is not possible to be notified when a child is stopped by a signal (such as SIGSTOP or SIGTTIN) or when a stopped child is resumed by delivery of a SIGCONT signal.
Monitoring child processes: waitpid()

- prototype of `waitpid()`:

```c
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *status, int options);
```

Returns process ID of child, 0, or –1 on error

- The `pid` argument enables the selection of the child to be waited for:
  - If `pid` is greater than 0, wait for the child whose process ID equals `pid`.
  - If `pid` equals 0, wait for any child in the same process group as the caller (parent)
  - If `pid` is less than –1, wait for any child whose process group identifier equals the absolute value of `pid`.
  - If `pid` equals –1, wait for any child. The call `wait(&status)` is equivalent to the call `waitpid(–1, &status, 0)`.
Monitoring child processes: waitpid()

- The **options** argument is a bit mask that can include (OR) zero or more of the following flags:
  - **WUNTRACED** - in addition to returning information about terminated children, also return information when a child is stopped by a signal.
  - **WCONTINUED** - return status information about stopped children that have been resumed by delivery of a SIGCONT signal.
  - **WNOHANG** - if no child specified by pid has yet changed state, then return immediately, instead of blocking. In this case, the return value of waitpid() is 0. If the calling process has no children that match the specification in pid, waitpid() fails with the error ECHILD.
Monitoring child processes: waitpid()

Figure: Value returned in the status argument of wait() and waitpid() [1]
Orphans and Zombies

- The orphaned child is adopted by init, the ancestor of all processes, whose process ID is 1.
  - after a child’s parent terminates, a call to getppid() will return the value 1. His can be used as a way of determining if a child’s true parent is still alive
- A child can terminate before its parent has had a chance to perform a wait()
  - kernel deals with this situation by turning the child into a zombie.
  - most of the resources held by the child are released back to the system to be reused by other processes. The only part of the process that remains is an entry in the kernel’s process table recording (among other things) the child’s process ID, termination status, and resource usage statistics
  - a zombie process can’t be killed by a signal, not even the SIGKILL. This ensures that the parent can always eventually perform a wait().
  - when the parent does perform a wait(), the kernel removes the zombie, since the last remaining information about the child is no longer required. On the other hand, if the parent terminates without doing a wait(), then the init process adopts the child and automatically performs a wait(), thus removing the zombie process from the system.
The SIGCHLD Signal

- The termination of a child process is an event that occurs asynchronously. A parent can’t predict when one of its child will terminate.

- Parent can use wait() or waitpid() in two ways:
  - without specifying the WNOHANG flag - the call will block if a child has not already terminated. But do we want the parent to be blocked?
  - with the WNOHANG flag - periodically perform a nonblocking check (a poll) for dead children. This solution wastes CPU time.

- Third option: handler for the **SIGCHLD signal**.

- **The SIGCHLD signal** is sent to a parent process whenever one of its children terminates.
  - By default, this signal is ignored, but can be caught by installing a signal handler.
  - within the signal handler, wait() (or similar) can be used to reap the zombie child.
The SIGCHLD Signal

```
#include <stdio.h>
#include <signal.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void handler(int sig)
{
    pid_t pid;
    pid = wait(NULL);
    printf("Pid %d exited.\n", pid);
}

int main(void)
{
    signal(SIGCHLD, handler);
    if(!fork())
    {
        printf("Child pid is %d\n", getpid());
        exit(0);
    }
    printf("Parent pid is %d\n", getpid());
    getchar();
    return 0;
}
```

Process execution

- The **execve() system call** loads a new program into a process's memory.
- During this operation, the old program is discarded, and the process’s stack, data, and heap are replaced by those of the new program.
- After executing various C library run-time startup code and program initialization code, the new program commences execution at its main() function.
- The most frequent use of execve() is in the child produced by a fork(), although it is also occasionally used in applications without a preceding fork().
Process execution

▶ Prototype of `execve()`:

```c
#include <unistd.h>
int execve(const char *pathname, char *const argv[],
            char *const envp[]);
```

Never returns on success; returns –1 on error

▶ `pathname` argument contains the pathname of the new program to be loaded into the process’s memory. It can be absolute (indicated by an initial `/`) or relative to the current working directory of the calling process.

▶ `argv` argument specifies the command-line arguments to be passed to the new program. This array corresponds to the second (argv) argument to a C main() function; it is a NULL-terminated list of pointers to character strings. argv[0] corresponds to the command name.

▶ `envp` argument specifies the environment list for the new program. It is a NULL-terminated list of pointers to character strings of the form name=value
Process execution

- Since `execve()` replaces the program that called it, a **successful execve() never returns**.
- If return value is -1, an error occurred. To determine the cause `errno` can be used.
- Among the errors that may be returned in `errno` are the following:
  - **EACCES** - the pathname argument doesn’t refer to a regular file, the file doesn’t have execute permission enabled, or one of the directory components of pathname is not searchable.
  - **ENOENT** - the file referred to by pathname doesn’t exist.
  - **ENOEXEC** - the file referred to by pathname is marked as being executable, but it is not in a recognizable executable format. Possibly, it is a script that doesn’t begin with a line (starting with the characters `#!`) specifying a script interpreter.
  - **ETXTBSY** - the file referred to by pathname is open for writing by another process.
  - **E2BIG** - the total space required by the argument list and environment list exceeds the allowed maximum.
Process execution - exec() function

The exec() library functions:

```c
#include <unistd.h>
int execl(const char *pathname, const char *arg, (char *) NULL, char *const envp[]);
int execlp(const char *filename, const char *arg, (char *) NULL);
int execvp(const char *filename, char *const argv[]);
int execv(const char *pathname, char *const argv[]);
int execl(const char *pathname, const char *arg, (char *) NULL);
```
Process execution - exec() function

- **execvp()** and **execvp()** allow the program to be specified using just a filename. The filename is sought in the list of directories specified in the PATH environment variable.

- **execlp(), execvp(), and execl()** require the programmer to specify the arguments as a list of strings within the call.

- **execve()** and **execle()** functions allow the programmer to explicitly specify the environment for the new program using envp, a NULL-terminated array of pointers to character strings.
Threads

- Like processes, threads are a mechanism that permits an application to perform multiple tasks concurrently.
- A single process can contain multiple threads.
- Threads share the same global memory, including the initialized data, uninitialized data, and heap segments.
- The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute parallel.
Threads versus Processes

▶ Processes:
  ▶ It is difficult to share information between processes. Since the parent and child don’t share memory (other than the read-only text segment), some form of interprocess communication in order to exchange information between processes has to be implemented.
  ▶ Process creation with fork() is relatively expensive. Even with the copy-on-write technique the need to duplicate various process attributes such as page tables and file descriptor tables means that a fork() call is still time-consuming.

▶ Threads:
  ▶ Sharing information between threads is easy and fast. It is just a matter of copying data into shared (global or heap) variables.
  ▶ Thread creation is faster than process creation—typically, ten times faster or better.
    ▶ Many of the attributes that must be duplicated in a child created by fork() are instead shared between threads.
    ▶ In particular, copy-on-write duplication of pages of memory is not required, nor is duplication of page tables.
Attributes shared by threads:

- process ID and parent process ID;
- process group ID and session ID;
- controlling terminal;
- open file descriptors;
- file system–related information: umask, current working directory, and root directory;
- interval timers (setitimer()) and POSIX timers (timer_create());
- resource limits;
- CPU time consumed (as returned by times());
- resources consumed (as returned by getrusage());
Threads

Attributes not shared by threads:

- thread ID;
- thread-specific data;
- the errno variable;
- stack (local variables and function call linkage information).
In the late 1980s and early 1990s, several different threading APIs existed. In 1995, POSIX.1c standardized the POSIX threads API, and this standard was later incorporated into SUSv3. The Pthreads API defines a number of data types. SUSv3 doesn’t specify how these data types should be represented, and portable programs should treat them as opaque data.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pthread_t</td>
<td>Thread identifier</td>
</tr>
<tr>
<td>pthread_mutex_t</td>
<td>Mutex</td>
</tr>
<tr>
<td>pthread_mutexattr_t</td>
<td>Mutex attributes object</td>
</tr>
<tr>
<td>pthread_cond_t</td>
<td>Condition variable</td>
</tr>
<tr>
<td>pthread_condattr_t</td>
<td>Condition variable attributes object</td>
</tr>
<tr>
<td>pthread_key_t</td>
<td>Key for thread-specific data</td>
</tr>
<tr>
<td>pthread_once_t</td>
<td>One-time initialization control context</td>
</tr>
<tr>
<td>pthread_attr_t</td>
<td>Thread attributes object</td>
</tr>
</tbody>
</table>

Figure: Pthreads data types [1]
The traditional method of returning status from system calls and some library functions is to return 0 on success and −1 on error, with errno being set to indicate the error.

All Pthreads functions return 0 on success or a positive value on failure. The failure value is one of the same values that can be placed in errno by traditional UNIX system calls.

```c
pthread_t *thread;
int s;

s = pthread_create(&thread, NULL, func, &arg);
if (s != 0)
    errExitEN(s, "pthread_create");
```
Pthreads API

- On Linux, programs that use the Pthreads API must be compiled with the `cc -pthread` option. The effects of this option include the following:
  - The `_REENTRANT` preprocessor macro is defined. This causes the declarations of a few reentrant functions to be exposed.
  - The program is linked with the `libpthread` library (the equivalent of `-lpthread`).
Pthreads API: Thread Creation

- **pthread_create()** function creates a new thread.
- **Prototype:**

  ```c
  #include <pthread.h>

  int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start)(void *), void *arg);
  ```

- Returns 0 on success, or a positive error number on error.
- **New thread** commences execution by calling the function identified by **start** with the argument **arg**.
- **Thread** argument points to a buffer of type pthread_t into which the unique identifier for this thread is copied before pthread_create() returns.
- **Attr** argument is a pointer to a pthread_attr_t object that specifies various attributes for the new thread.
- After a call to pthread_create(), a program has no guarantees about which thread will next be scheduled to use the CPU!
Pthreads API: Thread Termination

- The execution of a thread terminates in one of the following ways:
  - The thread’s start function performs a return specifying a return value for the thread.
  - The thread calls `pthread_exit()`
  - The thread is canceled using `pthread_cancel()`
  - Any of the threads calls `exit()`, or the main thread performs a `return` which causes all threads in the process to terminate immediately.

- `pthread_exit()` prototype:
  ```c
  include <pthread.h>
  
  void pthread_exit(void *retval);
  ```
  - `retval` can be obtained in another thread by calling `pthread_join()`.
Each thread within a process is uniquely identified by a thread ID.

This ID is returned to the caller of `pthread_create()`, and a thread can obtain its own ID using `pthread_self()`: 

```c
#include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

The `pthread_equal()` function allows to check whether two thread IDs are the same:

```c
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if `t1` and `t2` are equal, otherwise 0
Pthreads API: Joining Terminated Thread

The **pthread_join()** function **waits** for the thread identified by **thread** to **terminate** (If that thread has already terminated, pthread_join() returns immediately.)

```c
#include <pthread.h>

int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number on error

- If **retval** is a non-NULL pointer, then it receives a copy of the terminated thread’s return value—that is, the value that was specified when the thread performed a return or called pthread_exit().
Pthreads API: Joining Terminated Thread

- The `pthread_join()` function waits for the thread identified by `thread` to terminate (If that thread has already terminated, `pthread_join()` returns immediately.)

  ```
  include <pthread.h>
  
  int pthread_join(pthread_t thread, void **retval);
  ```

  Returns 0 on success, or a positive error number on error.

- If `retval` is a non-NULL pointer, then it receives a copy of the terminated thread’s return value—that is, the value that was specified when the thread performed a return or called `pthread_exit()`.

- **Calling `pthread_join()` for a thread ID that has been previously joined can lead to unpredictable behavior!**
Pthreads API: Joining Terminated Thread

```c
#include <pthread.h>

static void *threadFunc(void *arg) {
    char *s = (char *) arg;
    printf("%s", s);
    return (void *) strlen(s);
}

int main(int argc, char *argv[]) {
    pthread_t t1;
    void *res;
    int s;
    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0) {
        errExitEN(s, "pthread_create");
        printf("Message from main()\n");
        s = pthread_join(t1, &res);
        if (s != 0) {
            errExitEN(s, "pthread_join");
            printf("Thread returned %ld\n", (long) res);
            exit(EXIT_SUCCESS);
        }
    }
}```
By default, a thread is **joinable**, meaning that when it terminates, another thread can obtain its return status using `pthread_join()`.

Sometimes the thread’s return status is not important. In this case, the thread can be marked as **detached**, by making a call to `pthread_detach()` specifying the thread’s identifier in thread:

```c
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number on error

Thread can detach itself using the call: `pthread_detach(pthread_self());`

Once a thread has been detached, it is no longer possible to use `pthread_join()` to obtain its return status, and the thread can’t be made **joinable** again.
Bibliography