The exec Family and wait() Overview

A. exec Family

The exec functions completely replace the current process image with a new program. When a process calls an exec function, its entire code, data, and stack are overwritten by the new executable; only the PID (and some attributes like open file descriptors) remain unchanged. As a result, if the exec call is successful, nothing after it in your code gets executed. If there is an error, the exec call returns –1, and you can inspect `errno` for the error details.

Common exec variants include:

- **execl / execlp:**
  - *execl* takes a variable number of arguments (a list of strings terminated by a NULL pointer).
  - *execlp* works like execl but searches the PATH if the executable path isn't absolute.
- **execv / execvp:**
  - *execv* takes an array of pointers to character strings (the arguments), where the last pointer is NULL.
  - *execvp* also searches the PATH for the executable.
- **execve:**
  - Lower-level version that allows you to specify an environment variable array as well.

Key Points:

- Exec is normally used in a child process right after fork().
- It "upgrades" the child: its PID remains the same, but its code is replaced.
- Because exec never returns on success, any error handling (with a call to perror(), for example) should follow immediately after an exec call.

B. wait() and waitpid()

Once you've forked one or more processes that invoke exec, the parent may need to wait for its child(ren) to finish before continuing. The **wait()** or **waitpid()** functions enable the parent process to suspend its execution until a child terminates or changes state.

- **wait(int status):*
  Waits for any child process to terminate and returns the PID of that child.
- **waitpid(pid, int status, options):*
  Provides more control by letting you wait for a specific child or use options (like non-blocking wait).

Key Points:

- The integer pointed to by the status argument can be inspected using macros like `WIFEXITED(status)` and `WEXITSTATUS(status)` to determine the termination status of the child.
- Waiting prevents the child processes from becoming “zombies” (terminated processes still occupying a slot in the process table).

Code Example: Using execvp() and wait()

Below is a simple example that illustrates:

1. Forking a child process.
2. In the child, using execvp() to run a new program (e.g., listing directory contents with "ls -l").
3. In the parent, using wait() to suspend execution until the child finishes.

c
Copy

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

int main() {
    pid_t pid = fork(); // Create a child process.

    if (pid < 0) { // Check for fork() errors.
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        // Child process block.
        // Define the argument list for the new program.
        char *argv[] = {"ls", "-l", NULL};

        // Execute "ls -l" using execvp(). The execvp() function
searches for the executable "ls" in the PATH.
        if (execvp("ls", argv) == -1) {
```

```
            // If execvp returns, there was an error.
            perror("execvp failed");
            exit(EXIT_FAILURE);
        }
        // Note: No code beyond the execvp() call is executed if
execvp is successful.
    }
    else {
        // Parent process block.
        int status;
        // Wait for the child process to finish. wait() returns the
PID of the child that terminated.
        if (wait(&status) == -1) {
            perror("wait failed");
            exit(EXIT_FAILURE);
        }

        // Check if the child terminated normally.
        if (WIFEXITED(status)) {
            int exit_status = WEXITSTATUS(status);
            printf("Child terminated normally with exit status %d\n",
exit_status);
        }
        else {
            printf("Child did not terminate normally\n");
        }
    }

    return 0;
}
```

Explanation (Line-by-Line)

1. **Forking:**
    - `pid_t pid = fork();`
      The parent process creates a child. After the fork, two processes run concurrently.
2. **Error Checking:**

- If `pid < 0`, an error occurred, and the process prints an error message using `perror()` and exits with a failure status.

3. **Child Process (pid == 0):**
   - Define an array `argv` containing the program name `"ls"` and its argument `"-l"`, terminated by `NULL`.
   - The call `execvp("ls", argv);` replaces the child's current process image with that of the "ls" command. If successful, the code after execvp is not executed; if it returns, the call failed.
   - Error handling using `perror()` occurs if execvp fails.
4. **Parent Process (pid > 0):**
   - The parent calls `wait(&status);` to suspend execution until the child process terminates.
   - The status value is checked using `WIFEXITED(status)` to see if the child terminated normally.
   - `WEXITSTATUS(status)` is used to retrieve the return code from the child.
   - The parent then prints the exit status.

Additional Example: Using waitpid() for More Control

Sometimes you may want to wait for a specific child (or use non-blocking waits). The following example demonstrates the use of waitpid():

c
Copy

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

int main() {
    pid_t pid = fork();

    if (pid < 0) {
        perror("fork failed");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        // Child process.
```

```
        char *argv[] = {"ls", "-l", NULL};
        if (execvp("ls", argv) == -1) {
            perror("execvp failed");
            exit(EXIT_FAILURE);
        }
    }
    else {
        // Parent process waits specifically for the child with PID =
pid.
        int status;
        if (waitpid(pid, &status, 0) == -1) {
            perror("waitpid failed");
            exit(EXIT_FAILURE);
        }

        if (WIFEXITED(status)) {
            printf("Child %d exited with status %d\n", pid,
WEXITSTATUS(status));
        } else {
            printf("Child %d did not exit normally\n", pid);
        }
    }
    return 0;
}
```

Explanation

- **waitpid(pid, &status, 0):**
  Waits for the specific child identified by `pid`. This call gives more control, such as setting options (if needed) for non-blocking behavior.

Summary

- **exec Functions:**
  Replace the current process image with a new program. Use one of the exec variants (execl, execv, execlp, execvp, execve) depending on whether you have a variable-length

list of arguments or an array and whether you want the PATH to be searched automatically.

- **wait / waitpid:**
  Allow the parent process to synchronize with its child(ren) by waiting until they finish.
  - Use **wait()** to block until any child process terminates.
  - Use **waitpid()** for finer control over which child to wait for and to set options like non-blocking waits.

These concepts are crucial for building robust multi-process programs, where you create new processes (using fork and exec) and coordinate their termination (using wait or waitpid). Enjoy experimenting with these functions to build more complex systems!

Standard C Printing: printf() and Format Specifiers

A. printf() Overview

- **Function:**
  `printf()` prints formatted output to standard output (stdout).
- **Header:** `<stdio.h>`
- **Usage:**
  It uses a format string that may contain text mixed with format specifiers (tags) that tell the function how to interpret and format subsequent arguments.

B. Common Format Specifiers (Tags)

Specifier	Data Type	Description	Example Output
`%d` or `%i`	int	Signed decimal integer	42, -17
`%u`	unsigned int	Unsigned decimal integer	42
`%f`	double	Floating-point number	3.1416
`%lf`	double	(same as %f, but for clarity in scanf)	2.7183
`%c`	char	Single character	'A', 'z'
`%s`	char *	String	"Hello, world!"
`%p`	void *	Pointer address	0x7ffeefbff5c8
`%x` / `%X`	unsigned int	Unsigned hexadecimal integer	0x2a or 2A

%o	unsigned int	Unsigned octal	052
%e / %E	double	Scientific notation (lowercase/uppercase)	3.14e+00, 3.14E+00
%g / %G	double	Uses %f or %e (or %E) whichever is shorter	3.14

C. Code Example

c
Copy

```c
#include <stdio.h>

int main() {
    int num = 42;
    double pi = 3.14159;
    char ch = 'A';
    char *str = "Hello, world!";

    // Printing with printf:
    printf("Integer: %d\n", num);
    printf("Floating point: %f\n", pi);
    printf("Character: %c\n", ch);
    printf("String: %s\n", str);
    printf("Hexadecimal (lowercase): %x\n", num);
    printf("Pointer address: %p\n", (void *)str);

    // Combining multiple values
    printf("Combined: num=%d, pi=%f, ch=%c, str=%s\n", num, pi, ch,
str);

    return 0;
}
```

---

2. Simplified Output: puts() and putchar()

A. puts()

- **Function:**
  `puts()` prints a string to stdout and automatically appends a newline at the end.
- **Header:** `<stdio.h>`
- **Usage:**
  It is useful for quick printing when no formatting is needed.

B. Code Example: puts()

c
Copy

```c
#include <stdio.h>

int main() {
    puts("This is a simple string printed with puts.");
    // Note: No newline needed at the end of the string—puts adds one.
    return 0;
}
```

C. putchar()

- **Function:**
  `putchar()` prints a single character to stdout.
- **Usage Example:**

c
Copy

```c
#include <stdio.h>

int main() {
    putchar('H');
    putchar('i');
    putchar('\n');  // Adding a newline at the end.
    return 0;
}
```

---

3. Printing to Files: fprintf() and fputs()

A. fprintf() Overview

- **Function:**
  `fprintf()` prints formatted output to a specified stream (such as a file stream).
- **Header:** `<stdio.h>`
- **Usage:**
  Very similar to printf() but takes a file pointer as the first argument.

B. Code Example: fprintf()

c
Copy

```c
#include <stdio.h>
#include <stdlib.h>

int main() {
    FILE *fp = fopen("output.txt", "w");
    if (!fp) {
        perror("fopen failed");
        exit(EXIT_FAILURE);
    }
    fprintf(fp, "Writing an integer: %d, and a string: %s\n", 100,
"Hello, file!");
    fclose(fp);
    return 0;
}
```

C. fputs() Overview

- **Function:**
  `fputs()` writes a string to the specified stream.
- **Usage:**
  It does not add a newline automatically.

D. Code Example: fputs()

c
Copy

```c
#include <stdio.h>
#include <stdlib.h>
```

```c
int main() {
    FILE *fp = fopen("output.txt", "w");
    if (!fp) {
        perror("fopen failed");
        exit(EXIT_FAILURE);
    }
    fputs("This string is written with fputs.\n", fp);
    fclose(fp);
    return 0;
}
```

---

4. Printing to Strings: sprintf() and snprintf()

A. sprintf()

- **Function:**
  `sprintf()` writes formatted output to a string (char array), similar to printf.
- **Usage:**
  Use with caution regarding buffer size since it does not perform bounds checking.

B. Code Example: sprintf()

c
Copy

```c
#include <stdio.h>

int main() {
    char buffer[100];
    int num = 123;
    sprintf(buffer, "The number is: %d", num);
    printf("%s\n", buffer);
    return 0;
}
```

C. snprintf()

- **Function:**
  `snprintf()` works like sprintf(), but it includes a size parameter to avoid buffer overflows.
- **Usage:**
  Preferred over sprintf for safer string formatting.

D. Code Example: snprintf()

c
Copy

```c
#include <stdio.h>

int main() {
    char buffer[100];
    int num = 456;
    // Write to buffer, ensuring no more than 99 characters + null terminator.
    snprintf(buffer, sizeof(buffer), "The safe number is: %d", num);
    printf("%s\n", buffer);
    return 0;
}
```

---

5. Low-Level Output: write() and dprintf()

A. write() Overview

- **Function:**
  The `write()` system call writes raw bytes to a file descriptor (not buffered and without format specifiers).
- **Header:** `<unistd.h>`
- **Usage:**
  You must specify the exact number of bytes to write.
- **Example File Descriptors:**
  - STDOUT: 1
  - STDERR: 2

B. Code Example: write()

c
Copy

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

int main() {
    char message[] = "Low-level output with write.\n";
    // Write to standard output (file descriptor 1):
    write(1, message, strlen(message));
    return 0;
}
```

C. dprintf() Overview

- **Function:**
  `dprintf()` behaves like printf() but writes its output directly to a given file descriptor.
- **Header:** `<stdio.h>` (POSIX)
- **Usage Example:**
  Useful when you wish to write formatted output to a file descriptor not associated with a standard stream.

D. Code Example: dprintf()

c
Copy

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

int main() {
    int fd = 1;  // For example, stdout.
    dprintf(fd, "dprintf prints a formatted number: %d\n", 789);
    return 0;
}
```

---

6. Summary of Print-Related Functions and Their “Tags”

- **printf / fprintf / dprintf:**
  Use format strings (tags) like `%d`, `%s`, `%f`, `%c`, `%x`, etc., to format data.

- **puts / fputs / putchar:**
  Simpler functions when no complex formatting is needed.
- **sprintf / snprintf:**
  Write formatted output to a string; use snprintf for safer, bounded writes.
- **write:**
  Write raw bytes to a file descriptor with no formatting support.

When to Use Each Method

- **For User-Facing Console Output:**
  Use `printf()` or `puts()` for simplicity and formatted output.
- **When Writing to Files:**
  Use `fprintf()` with a FILE pointer (obtained via fopen()).
- **When Working with File Descriptors Directly:**
  Use `dprintf()` for formatted output or `write()` for raw binary data.
- **For Constructing Strings Internally:**
  Use `sprintf()` or `snprintf()` to build strings for later use.

Additional Notes on Formatting Tags

- **Width, Precision, and Flags:**
  You can modify specifiers with flags, field width, and precision. For example:
  - `%5d` prints an integer in a field width of 5.
  - `%-5d` left-justifies the integer.
  - `%.2f` prints a floating-point number with 2 decimals.
  - `%06d` prints an integer padded with zeros to 6 digits.

Example:

c
Copy

```c
#include <stdio.h>

int main() {
    int num = 42;
    double pi = 3.14159;
    printf("Padded number: %06d\n", num);
```

```
    printf("Floating point (2 decimals): %.2f\n", pi);
    return 0;
}
```

Conclusion

This note sheet provides a comprehensive overview of all the different printing methods available in C. Whether you're printing to the screen, formatting output into a string, writing directly to file descriptors, or using low-level system calls, understanding the differences and proper usage is key to writing effective and safe C programs. Use these examples and guidelines as a quick reference or cheat sheet for your systems programming assignments. Happy coding!