Unix Processes: Wait and Signal API

Kishan Kumar Ganguly
Asst. Prof.
IIT, DU

Understanding Zombie Processes in Unix

- Child processes don't disappear immediately when terminated.
- Instead, they enter a special process state called a **zombie state**.
- In this state, only minimal kernel data structures are retained.
- Zombie processes wait for their parent to inquire about their status.
- This waiting process is known as **waiting on the zombie process**.
- Only after the parent obtains information about the terminated child, the process formally exits, even as a zombie.

Demonstrating Zombie Processes in C Code

```c
int main() {
    pid_t child_pid;
    // Create child process
    child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process is running.\n");
        sleep(2); // Simulate some work
        printf("Child process is done.\n");
        exit(0);
    }
    else if (child_pid > 0) {
        // Parent process
        printf("Parent process created child with PID: %d\n",child_pid);
        sleep(4); // Parent sleeps for a while
        printf("Parent process woke up.\n");
    } else {
        // Error handling
        perror("fork");
        exit(1);
    }
    // Parent process doesn't wait for child
    printf("Parent process is done.\n");
    return 0;
}
```

Demonstrating Zombie Processes in C Code

```c
int main() {
    pid_t child_pid;
    // Create child process
    child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process is running.\n");
        sleep(2); // Simulate some work
        printf("Child process is done.\n");
        exit(0);
    }
```

Zombie processes exist until the parent collects their exit status using `wait()`

```c
    else if (child_pid > 0) {
        // Parent process
        printf("Parent process created child with PID: %d\n",child_pid);
        sleep(4); // Parent sleeps for a while
        printf("Parent process woke up.\n");
        // Parent collects child's exit status, removing the zombie

        int status;
        pid_t terminated_pid;

        while ((terminated_pid = wait(&status)) > 0) {
            if (WIFEXITED(status))
                printf("Parent: Child process (PID: %d) terminated with status
%d\n", terminated_pid, WEXITSTATUS(status));
            else
                printf("Parent: Child process (PID: %d) terminated
abnormally\n", terminated_pid);
        }
    }
    .....
    return 0;
}
```

Wait for Terminated Child Process

- **wait()** retrieves the PID of a terminated child process or returns -1 on error.
- Blocks if no child has terminated, returns immediately if a child has.
- Useful for responding to child termination events (e.g., upon receiving a **SIGCHLD** signal).
- Possible error codes include **ECHILD (no children)** and **EINTR (interrupted by a signal)**.

Wait for Terminated Child Process

```c
int main() {
    pid_t child_pid;
    int status;
    printf("Parent process (PID: %d) is running.\n", getpid());
    child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process (PID: %d) is running.\n", getpid());
        exit(42);  // Simulate exit with status 42
    }
```

```c
    else if (child_pid > 0) {
        // Parent process
        wait(&status);  // Wait for the child to terminate
        if (WIFEXITED(status)) {
            printf("Child process (PID: %d) exited normally with status: %d\n", child_pid, WEXITSTATUS(status));
        }
    } else {
        perror("fork");
        exit(1);
    }
    printf("Parent process is done.\n");
    return 0;
}
```

Checking Different Status Macros

```c
int main() {

    pid_t child_pid;
    int status;

    printf("Parent process (PID: %d) is running.\n", getpid());

    child_pid = fork();

    if (child_pid == 0) {

        // Child process

        printf("Child process (PID: %d) is running.\n", getpid());

        exit(42);  // Simulate exit with status 42

    } else if (child_pid > 0) {

        // Parent process
        wait(&status);  // Wait for the child to terminate

        if (WIFEXITED(status)) {

            printf("Child process (PID: %d) exited normally with status: %d\n", child_pid, WEXITSTATUS(status));
        } else if (WIFSIGNALED(status)) {

            printf("Child process (PID: %d) terminated due to signal: %d\n", child_pid, WTERMSIG(status));
            if (WCOREDUMP(status)) {

                printf("Core dumped by child process.\n");
            }
        }
    } else {
        perror("fork");
        exit(1);
    }
    printf("Parent process is done.\n");
    return 0;
}
```

Checking Process Stopped and Continued Status

- WIFSTOPPED and WIFCONTINUED can be used to check if a child process was stopped or continued
- Demonstrated later

Waitpid() vs. Wait()

- **wait()** can be cumbersome when dealing with multiple children.
- **waitpid()** provides more control and flexibility.

Using waitpid()

- **waitpid()** is a powerful version of wait() with additional parameters for fine-tuning
- **Parameters:**
    - pid: Specifies the exact process or processes to wait for.
    - status: Works the same way as wait().
    - options: Allows for customization.
- **pid Values:**
    - < -1: Wait for any child in a specific process group.
    - -1: Wait for any child (same as wait()).
    - 0: Wait for any child in the same process group as the caller.
    - > 0: Wait for a child with a specific PID.

Options Parameter in waitpid()

- **options:** A binary OR of options:
  WNOHANG: Return immediately if no matching child has terminated.
  WUNTRACED: Set WIFSTOPPED bit even if not tracing the child.
  WCONTINUED: Set WIFCONTINUED bit even if not tracing the child.
- **Return Values:**
  On success, waitpid() returns the PID of the changed process.
  If WNOHANG specified and no change, it returns 0.
  On error, it returns -1.
- **Error Codes:**
  ECHILD: Specified process(es) do not exist or aren't children.
  EINTR: Signal received while waiting (if WNOHANG not used).
  EINVAL: Invalid options argument.

Code Example with waitpid()

```c
int main() {
    pid_t child_pid = fork();
    if (child_pid == 0) {
        // Child process
        printf("Child process (PID: %d) is running.\n", getpid());
        sleep(2); // Simulate some work
        exit(42); // Exit with status 42
    } else if (child_pid > 0) {
        // Parent process
        printf("Parent process (PID: %d) is running.\n", getpid());
        int status;
        // Use waitpid to wait for the specific child (PID 1742)
        pid_t specific_pid = 1742;
        pid_t terminated_pid = waitpid(specific_pid, &status, WNOHANG);
        if (terminated_pid == specific_pid) {
            if (WIFEXITED(status))
                printf("Child process (PID: %d) exited with status: %d\n", terminated_pid, WEXITSTATUS(status));
            else  printf("Child process (PID: %d) terminated abnormally.\n", terminated_pid);
        } else if (terminated_pid == 0) {
            printf("Child process (PID: %d) has not yet terminated.\n", specific_pid);
        } else
            perror("waitpid");
    } else {
        perror("fork");
        exit(1);
    }
    return 0;
}
```

Introduction to Signaling in C Processes

- Processes can send signals to each other or themselves.
- Signals are software interrupts notifying a process about an event.
- Common Signals:
    - SIGINT: Interrupt from the keyboard (e.g., Ctrl+C).
    - SIGTERM: Termination request.
    - SIGKILL: Forceful termination.
    - SIGSTOP: Pause execution
    - SIGUSR1 and SIGUSR2: User-defined signals.
- Each signal has a unique number.

Sending Signals

- Processes can send signals using **kill()** or **raise()** functions.
- **kill(pid, signal)**: Send a signal to a specific process.
- **raise(signal)**: Send a signal to the current process.

Sending Signals

- Processes can send signals using **kill()** or **raise()** functions.
- **kill(pid, signal)**: Send a signal to a specific process.
- **raise(signal)**: Send a signal to the current process.

Example - Sending Signals

```c
int main() {
    pid_t pid = getpid();
    printf("My PID: %d\n", pid);
    // Send SIGUSR1 to the current process
    raise(SIGUSR1);
    sleep(2); // Allow time for signal handling
    return 0;
}
```

Example - Sending Signals

```c
int main() {
    pid_t target_pid = 12345;  // Replace with the actual PID of the target process
    int signal_type = SIGUSR1; // Specify the signal type (e.g., SIGUSR1)
    int result = kill(target_pid, signal_type);
    if (result == 0) {
        printf("Signal %d sent to process with PID %d.\n", signal_type, target_pid);
    } else {
        perror("kill");
    }
    return 0;
}
```

Handling Signals

- Processes can define signal handlers using signal() or sigaction().
- Signal handlers specify how the process should respond to signals.

Example - Handling SIGINT

```c
void sigint_handler(int signum) {
    printf("Received SIGINT (Ctrl+C).\n");
}
int main() {
    signal(SIGINT, sigint_handler); // Register SIGINT handler
    while (1) {}
    return 0;
}
```

Classwork

- Use wait and signaling techniques to demonstrate **WIFSTOPPED(status)** and **WIFCONTINUED(status)**

Classwork

https://snipit.io/public/lists/24733/82644

Thank You!