Exceptional Control Flow

Topics for HW4

(Slides adapted from CSAPP)
Reviewing Linux system call for process control

**wait**: Synchronizing with Children

- **zombie**: When process terminates, still consumes system resources. Various tables maintained by OS.

- **How to reap/synchronize children?**
  - `int wait(int *child_status)`
    - suspends current process until one of its children terminates
    - return value is the **pid** of the child process that terminated
    - if `child_status != NULL`, then the object it points to will be set to a status indicating why the child process terminated
  - `waitpid`: suspends current process until a particular children terminates
  - **Wait Macros**
    - WEXITSTATUS: Returns the exit status of a normally terminated child
    - WIFEXITED : Returns true if the child terminated normally (via a call to exit or return)
**Example** (using `wait` for synchronizing Child processes and parent and checking the Child processes' status)

- Use fork to parallelize the summation over an array

```c
pid_t pid[2];
if ((pid[0] = fork()) == 0) { /* Child runs user job */
    for(i=0; i<5; i++)
        sum = sum + a[i];
    exit(sum);
} else {
    if ((pid[1] = fork()) == 0) { /* Child runs user job */
        for(i=5; i<10; i++)
            sum = sum + a[i];
        exit(sum);
    } else {
        int child_status;
        for (i = 1; i >= 0; i--)
            pid_t wpid = waitpid(pid[i], &child_status, 0);
        if (WIFEXITED(child_status)) {
            sum = sum + WEXITSTATUS(child_status);
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        } else
            printf("Child %d terminated abnormally\n", wpid);
    }
    printf("Parent process: The sum of array is:%d\n", sum);
}
```

How can child communicate with the parent?
- exit status
- shared memory
- files
...

First child to the sum on the first part of the array

Second child to the sum on the second part of the array

Collect the results and calculate the final result (child pass their results by their status)

Sample input: 1 1 1 1 1 2 2 2 2
Sample output:
Child 18005 terminated with exit status 10
Child 18004 terminated with exit status 5
Parent process: The sum of array is: 15
execve: Loading and Running Programs

- int execve(
  char *filename,
  char *argv[],
  char *envp[]
)

- Loads and runs in current process:
  - Executable `filename`
  - With argument list `argv`
  - And environment variable list `envp`

- Does not return (unless error)

- Overwrites code, data, and stack
  - keeps pid, open files and signal context

- Environment variables:
  - “name=value” strings
  - getenv and putenv
main()
Int child_status;
if ((pid = Fork()) == 0) { /* Child runs user job */
    if (execve(argv[0], argv, environ) < 0) {
        printf("%s: Command not found.\n", argv[0]);
        exit(0);
    }
}
wait(&child_status);

argv[argc] = NULL
argv[argc-1]
...
argv[0]

"/usr/include"
"-lt"
"ls"

envp[n] = NULL
envp[n-1]
...
envp[0]

"PWD=/usr/droh"
"PRINTER=iron"
"USER=droh"
Programmer’s Model of Multitasking

- **Basic functions**
  - `fork` spawns new process
    - Called once, returns twice
  - `exit` terminates own process
    - Called once, never returns
    - Puts it into “zombie” status
  - `wait` and `waitpid` wait for and reap terminated children
  - `execve` runs new program in existing process
    - Called once, (normally) never returns

- **Programming challenge**
  - Understanding the nonstandard semantics of the functions
  - Avoiding improper use of system resources
    - E.g. “Fork bombs” can disable a system
Shell Programs

- A shell is an application program that runs programs on behalf of the user.
  - **sh** Original Unix shell (Stephen Bourne, AT&T Bell Labs, 1977)
  - **csh** BSD Unix C shell (**tcsh**: enhanced **csh** at CMU and elsewhere)
  - **bash** “Bourne-Again” Shell

```c
int main() {
    char cmdline[MAXLINE];

    while (1) {
        /* read */
        printf("> ");
        fgets(cmdline, MAXLINE, stdin);
        if (feof(stdin))
            exit(0);

        /* evaluate */
        eval(cmdline);
    }
}
```

Execution is a sequence of read/evaluate steps.
void eval(char *cmdline) {
    char *argv[MAXARGS];  /* argv for execve() */
    int bg;               /* should the job run in bg or fg? */
    pid_t pid;            /* process id */

    bg = parsline(cmdline, argv);  //returns 1 if it is a background job
    if (!builtin_command(argv)) {
        if ((pid = fork()) == 0) {  /* child runs user job */
            if (execve(argv[0], argv, environ) < 0) {
                printf("%s: Command not found.\n", argv[0]);
                exit(0);
            }
        }
    }

    if (!bg) {  /* parent waits for fg job to terminate */
        int status;
        if (waitpid(pid, &status, 0) < 0)
            unix_error("waitfg: waitpid error");
    }
    else  /* otherwise, don’t wait for bg job */
        printf("%d %s", pid, cmdline);
}

What Is a “Background Job”?

- Users generally run one command at a time
  - Type command, read output, type another command

- Some programs run “for a long time”
  - Example: “delete this file in two hours”
    ```
    unix> sleep 7200; rm /tmp/junk  # shell stuck for 2 hours
    ```

- A “background” job is a process we don't want to wait for
  ```
  unix> (sleep 7200 ; rm /tmp/junk) &
  [1] 907
  unix> # ready for next command
  ```
Problem with Simple Shell Example

- Our example shell correctly waits for and reaps foreground jobs

- But what about background jobs?
  - Will become zombies when they terminate
  - Will never be reaped because shell (typically) will not terminate
  - Will create a memory leak that could run the kernel out of memory
Exceptional Control Flow (ECF) to the Rescue!

- Problem
  - The shell doesn't know when a background job will finish
  - By nature, it could happen at any time
  - The shell's regular control flow can't reap exited background processes in a timely fashion
  - Regular control flow is “wait until running job completes, then reap it”

- Solution: Exceptional control flow
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the alert mechanism is called a *signal*
ECF Exists at All Levels of a System

- Exceptions
  - Hardware and operating system kernel software
- Process Context Switch
  - Hardware timer and kernel software
- Signals
  - Kernel software
- Nonlocal jumps
  - Application code
A signal is a small message that notifies a process that an event of some type has occurred in the system

- akin to exceptions and interrupts
- sent from the kernel (sometimes at the request of another process) to a process
- signal type is identified by small integer ID’s (1-30)
- only information in a signal is its ID and the fact that it arrived

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Default Action</th>
<th>Corresponding Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>SIGINT</td>
<td>Terminate</td>
<td>Interrupt (e.g., ctl-c from keyboard)</td>
</tr>
<tr>
<td>9</td>
<td>SIGKILL</td>
<td>Terminate</td>
<td>Kill program (cannot override or ignore)</td>
</tr>
<tr>
<td>11</td>
<td>SIGSEGV</td>
<td>Terminate &amp; Dump</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>14</td>
<td>SIGALRM</td>
<td>Terminate</td>
<td>Timer signal</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Signal handling

- An exception is a transfer of control to the OS in response to some event (i.e., change in processor state)

Examples:
- Kernel has detected a system event such as divide-by-zero (SIGFPE) or the termination of a child process (SIGCHLD)
- Another process has invoked the `kill`
Pending and Blocked Signals

- A signal is *pending* if sent but not yet received
  - There can be at most one pending signal of any particular type
  - Important: Signals are not queued
    - If a process has a pending signal of type k, then subsequent signals of type k that are sent to that process are discarded

- A process can *block* the receipt of certain signals
  - Blocked signals can be delivered, but will not be received until the signal is unblocked

- A pending signal is received at most once
Default Actions

- Each signal type has a predefined default action, which is one of:
  - The process terminates
  - The process terminates and dumps core
  - The process stops until restarted by a SIGCONT signal
  - The process ignores the signal
Installing Signal Handlers

- The `signal` function modifies the default action associated with the receipt of signal `signum`:
  - `handler_t *signal(int signum, handler_t *handler)`

- Different values for `handler`:
  - SIG_IGN: ignore signals of type `signum`
  - SIG_DFL: revert to the default action on receipt of signals of type `signum`
  - Otherwise, `handler` is the address of a `signal handler`
    - Called when process receives signal of type `signum`
    - Referred to as “installing” the handler
    - Executing handler is called “catching” or “handling” the signal
    - When the handler executes its return statement, control passes back to instruction in the control flow of the process that was interrupted by receipt of the signal
**Signal Handling Example**

```c
void int_handler(int sig) {
    printf("Process %d received signal %d\n", getpid(), sig);
    exit(0);
}

void fork13() {
    pid_t pid[N];
    int i, child_status;
    signal(SIGINT, int_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            while(1); /* child infinite loop
    }  
    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", wpid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminated abnormally\n", wpid);
    }
}
```

---

`linux> ./forks 13`  
Killing process 25417  
Killing process 25418  
Killing process 25419  
Killing process 25420  
Process 25420 received signal 2  
Process 25421 received signal 2  
Child 25417 terminated with exit status 0  
Child 25418 terminated with exit status 0  
Child 25420 terminated with exit status 0  
Child 25419 terminated with exit status 0  
Child 25421 terminated with exit status 0  
`linux>`
Signals Handlers as Concurrent Flows

- A signal handler is a separate logical flow (not process) that runs concurrently with the main program
  - “concurrently” in the “not sequential” sense

```
Process A
while (1) {
    handler()
    ...
}

Process A

Process B
```
Signal Handler problem
use signal handler to reap children when they are terminated

```c
int ccount = 0;
void child_handler(int sig)
{
    int child_status;
    pid_t pid = wait(&child_status);
    ccount--;
    printf("Received signal %d from process %d\n", sig, pid);
}

void fork14()
{
    pid_t pid[N];
    int i, child_status;
    ccount = N;
    signal(SIGCHLD, child_handler);
    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            sleep(1); /* deschedule child */
            exit(0); /* Child: Exit */
        }
    while (ccount > 0)
        pause(); /* Suspend until signal occurs */
}
```

- Pending signals are not queued
  - For each signal type, just have single bit indicating whether or not signal is pending
  - Even if multiple processes have sent this signal

```
linux> ./forks 14
Received SIGCHLD signal 17 for process 21344
Received SIGCHLD signal 17 for process 21345
```
Living With Nonqueuing Signals

- Must check for all terminated jobs
  - Typically loop with `wait`

```c
void child_handler2(int sig)
{
    int child_status;
    pid_t pid;
    while ((pid = waitpid(-1, &child_status, WNOHANG)) > 0) {
        ccount--;
        printf("Received signal %d from process %d\n", sig, pid);
    }
}
```

```c
void fork15()
{
    . . .
    signal(SIGCHLD, child_handler2);
    . . .
}
```

greatwhite> forks 15
Received signal 17 from process 27476
Received signal 17 from process 27477
Received signal 17 from process 27478
Received signal 17 from process 27479
Received signal 17 from process 27480

greatwhite>
Summary

- Signals provide process-level exception handling
  - Can generate from user programs
  - Can define effect by declaring signal handler

- Some caveats
  - Very high overhead
    - >10,000 clock cycles
    - Only use for exceptional conditions
  - Don’t have queues
    - Just one bit for each pending signal type
Reviewing HW4

- **eval**: Main routine that parses and interprets the command line. [70 lines]
- **builtin cmd**: Recognizes and interprets the built-in commands: quit, fg, bg, and jobs. [25 lines]
- **do bgfg**: Implements the bg and fg built-in commands. [50 lines]
- **waitfg**: Waits for a foreground job to complete. [20 lines]
- **sigchld handler**: Catches SIGCHLD signals. 80 lines]
- **sigint handler**: Catches SIGINT (ctrl-c) signals. [15 lines]
- **sigtstp handler**: Catches SIGTSTP (ctrl-z) signals. [15 lines]