Process Control con’t
**wait()**: Synchronizing with Children

- Parent reaps a child by calling the `wait` function

- `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 integer it points to will be set to a value that indicates reason the child terminated and the exit status.
  - See textbook for more details.
```c
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("Child!");
        exit(0);
    } else {
        printf("Parent!");
        wait(&child_status);
        printf("CT!");
    }
    printf("Bye\n");
}
```

**Feasible output:**
- Parent!
- Child!
- CT!
- Bye

**Infeasible output:**
- Parent!
- CT!
- Bye
- Child!
Another `wait()` Example

- If multiple children completed, will take in arbitrary order
- Use `WIFEXITED` and `WEXITSTATUS` to get exit status

```c
void fork10() {
    int pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    for (i = 0; i < N; i++) { /* Parent */
        int child_pid = wait(&child_status);
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", child_pid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", child_pid);
    }
}
```
waitpid(): Waiting for a Specific Process

- int waitpid(pid_t pid, int &child_status, int options)
  - Suspends current process until specific process terminates
  - Various options (see textbook)

```c
void fork11() {
    int pid[N];
    int i, child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0)
            exit(100+i); /* Child */

    for (i = N-1; i >= 0; i--)
        if (WIFEXITED(child_status))
            printf("Child %d terminated with exit status %d\n", child_pid, WEXITSTATUS(child_status));
        else
            printf("Child %d terminate abnormally\n", child_pid);
}
```
**execve()** : Loading and Running Programs

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

- ** Loads and runs in the current process:**
  - Executable file `filename`
  - Argument list `argv`
  - Environment variable list `envp`
    - “name=value” strings (e.g., `USER=rshepherd`)
    - `getenv()`, `putenv()`, `printenv()`

- **Overwrites code, data, and stack**
  - Retains PID, open files and signal context

- **Called once and never returns**
  - …except if there is an error

- See book for more details.
Summary

- Creating processes
  - Call fork
  - One call, two returns

- Process completion
  - Call exit
  - One call, no return

- Reaping and waiting for processes
  - Call wait or waitpid

- Loading and running programs
  - Call execve
  - One call, (normally) no return
Signals
Linux Process Hierarchy

Note: you can view the hierarchy using the Linux `pstree` command
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)
- **bash**  
  “Bourne-Again” Shell (default Linux shell)
- **csh, zsh** ... many others

```c
int main()
{
    char cmdline[MAXLINE]; /* command line */

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

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

Execution is a sequence of read/evaluate steps
```c
void eval(char* cmdline)
{
    char* argv[MAXARGS];    /* Argument list for program to be run*/
    int bg;                  /* Should the job run in bg or fg? */
    int pid;                 /* Process id */

    bg = parseline(cmdline, argv);      /* Extract arguments and set bg */

    if ((pid = Fork()) == 0) {        /* Child runs user job */
        if (execve(argv[0], argv) < 0) {
            printf("%s: Command not found.\n", argv[0]);
            exit(0);
        }
    }

    /* Parent waits for foreground job to terminate */
    if (!bg) {
        int status;
        if (waitpid(pid, &status, 0) < 0)
            printf("waitfg: waitpid error %d", status);
    } else {
        printf("%d %s", pid, cmdline);
    }

    return;
}
```
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 (probably) will not terminate
  - Will create a memory leak that could run the kernel out of memory
Solution: Exceptional control flow

- We can leverage exceptional control flow from our programs
  - The kernel will interrupt regular processing to alert us when a background process completes
  - In Unix, the mechanism is called a signal
**Signals**

- 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)
  - Signal type is identified by 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>User typed ctrl-c</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</td>
<td>Segmentation violation</td>
</tr>
<tr>
<td>17</td>
<td>SIGCHLD</td>
<td>Ignore</td>
<td>Child stopped or terminated</td>
</tr>
</tbody>
</table>
Signal Concepts: Sending a Signal

- Kernel sends (delivers) a signal to a destination process by updating some state in the context of the destination process.

- Kernel sends a signal for one of the following reasons:
  - Kernel has detected a system event such as the termination of a child process (SIGCHLD).
  - Another process has invoked the `kill` system call to explicitly request the kernel to send a signal to the destination process.
A destination process receives a signal when it is forced by the kernel to react in some way to the delivery of the signal.

Some possible ways to react:

- *Ignore* the signal
- *Terminate* the process
- *Catch* signal by executing a user-level function called *signal handler*
  - Like exception handler called in response to an async interrupt

Signal Concepts: Receiving a Signal

1. Signal received by process
2. Control passes to signal handler
3. Signal handler runs
4. Signal handler returns to next instruction
Signal Concepts: Pending & Blocked

- 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
  - Cannot block SIGKILL or SIGSTOP

- A *pending* signal is received at most once
Signal Concepts: Pending/Blocked Bits

- Kernel maintains **pending** and **blocked** bit vectors in the context of each process
  - **pending**: represents the set of pending signals
    - Kernel sets bit k in pending when a signal of type k is delivered
    - Kernel clears bit k in pending when a signal of type k is received
  - **blocked**: represents the set of blocked signals
    - Can be set and cleared by using the `sigprocmask` function
    - Also sometimes referred to as the “signal mask”.
Sending Signals: Process Groups

- Every process belongs to exactly one process group

```plaintext
getpgid()
Return process group of current process

setpgid()
Change process group of a process (see text for details)
```
Sending Signals with /bin/kill Program

- /bin/kill program sends specified signal to a process or process group

Examples
- /bin/kill -9 24818
  Send SIGKILL to process 24818

- /bin/kill -9 -24817
  Send SIGKILL to every process in process group 24817
void fork12()
{
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
        if ((pid[i] = fork()) == 0) {
            /* Child: Infinite Loop */
            while(1) {}
        }

    for (i = 0; i < N; i++) {
        printf("Killing process %d\n", pid[i]);
        kill(pid[i], SIGINT);
    }

    for (i = 0; i < N; i++) {
        int 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);
    }
}
Receiving Signals

- Suppose kernel is returning from an exception handler and is ready to pass control to process $B$
Receiving Signals con’t

- Suppose kernel is returning from an exception handler and is ready to pass control to process B

- Kernel computes \( pnb = \text{pending} \& \sim \text{blocked} \)
  - The set of pending nonblocked signals for process B

- If \( (pnb == 0) \)
  - Pass control to next instruction in the logical flow for B

- Else
  - Choose nonzero bit \( k \) in \( pnb \) and force process B to receive signal \( k \)
  - The receipt of the signal triggers some action by B
  - Repeat for all nonzero bits in \( pnb \)
  - Pass control to next instruction in logical flow for B
Default Actions

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

- What if we do not like the default action?
  - 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 (macros for common cases):

- `SIG_IGN`: ignore signals of type `signum`
- `SIG_DFL`: revert to the default action for signals of type `signum`
- Otherwise, handler is the address of a user-level `signal handler`
  - Called when process receives signal of type `signum`
  - 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.

- Returns the previous value of the signal handler, or `SIG_ERR` on error.
void sigint_handler(int sig) /* SIGINT (ctrl+c) handler */ {
    printf("You want me to quit???
");
    sleep(2);
    exit();
}

int main() {
    /* Install the SIGINT handler */
    if (signal(SIGINT, sigint_handler) == SIG_ERR)
        unix_error("signal error");

    /* Wait for the receipt of a signal */
    pause();

    return 0;
}