CS 4400
Computer Systems

LECTURE 17

More on process control

Signals

Nonlocal jumps

Question – *Review* `fork()`

```
#include "csapp.h"

void doit() {
  if(Fork() == 0) {
    Fork();
    printf("hello\n");

    if(Fork() != 0)
      exit(0);
  }
}

int main() {
  doit();
  printf("hello\n");
  exit(0);
}
```

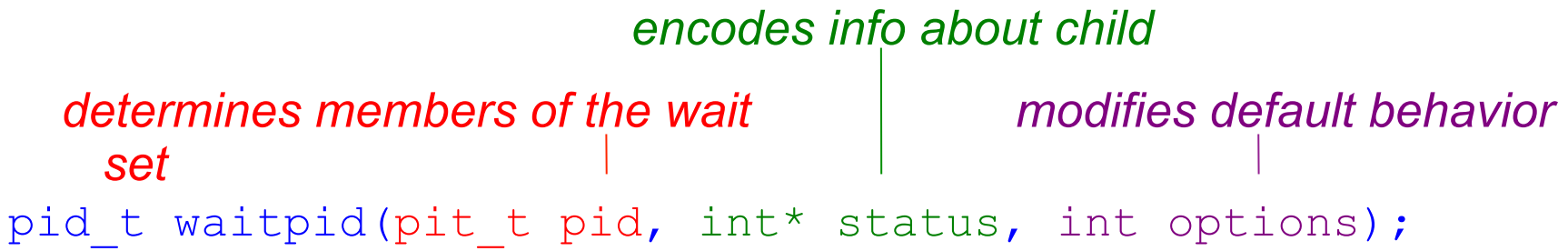
How many "hello" output lines does this program print?

Reaping Child Processes

- When a process terminates, the kernel does not remove it from the system immediately.
- The process is retained in a terminated state until it is *reaped* by its parent.
  - a terminated process not yet reaped is called a *zombie*
- If the parent terminates without reaping its children, the kernel arranges for the `init` process to reap them.
  - `init` has PID 1 and is created during system initialization
  - long running programs (i.e., shells) *should always* reap their zombie children because they consume system memory

`waitpid` Function

- A process waits for its children to terminate by calling



- By default, `waitpid` suspends execution of the calling process until a child process in its *wait set* terminates.
  - if a process in the wait set has already terminated, `waitpid` returns immediately
  - returns the PID of the terminated child causing `waitpid` to return
  - terminated child is then removed from the system

Determining the Wait Set

- If `pid` > `0`, then the wait set is the singleton child process whose PID is equal to `pid`. If `pid` = `-1`, then the wait set consists of all of the parent's child processes.
- Standard macros interpret the value of `status`.
  - `WIFEXITED(status)` is true if child terminated normally
  - `WIFEXITSTATUS(status)` returns exit status of child
  - see text for more macros
- If there are no children, `waitpid` returns -1 and `errno` set to `ECHILD`.
  - also returns -1 if interrupted by a signal (`errno` set to `EINTR`)

Example: `waitpid`

```
/* waitpid1.c */

#include "csapp.h"
#define N 2

int main() {
  int status, i;
  pid_t pid;

  for(i = 0; i < N; i++)
     if((pid = Fork()) == 0)  /* child */
       exit(100+i);

  /* parent waits for all of its children to terminate */
  while((pid = waitpid(-1, &status, 0)) > 0) {
     if(WIFEXITED(status))
       printf("child %d terminated normally with exit status=%d\n",
              pid, WEXITSTATUS(status));
    else
       printf("child %d terminated abnormally\n", pid);
  }
  if(errno != ECHILD)
    unix_error("waitpid error");

  exit(0);
}
```

Will the children always be reaped "in order"?

```
unix> ./waitpid1
child 22966 terminated normally with exit status=100
child 22967 terminated normally with exit status=101
```

Question

```
#include "csapp.h"

int main() {
  int status;
  pid_t pid;

  printf("Hello\n");
  pid = Fork();
  printf("%d\n", !pid);
  if(pid != 0)
    if(waitpid(-1, &status, 0) > 0)
      if(WIFEXITED(status) != 0)
        printf("%d\n", WEXITSTATUS(status));

  printf("Bye\n");
  exit(2);
}
```

What does this print? Is output the same every time?

Question

```
#include "csapp.h"

/* Wait() = Waitpid() with pid and options set to
   defaults; it blocks until any child terminates. */

int main() {
  if(Fork() == 0) {
    if(Fork() == 0)
      printf("a");
    else {
      pid_t pid; int status;
      if((pid = Wait(&status)) > 0)
        printf("b");
    }
  }
  else {
    printf("c");
    exit(0);
  }
  printf("d");
  return 0;
}
```

Which outputs are possible?

acdbd

adbdc

abddc

cadbd

bdadc

`sleep` and `pause`

- `sleep` suspends a process for some period of time.

```
unsigned int sleep(unsigned int secs);
```

  - returns 0 if the requested amount of time has already elapsed
  - otherwise, returns number of seconds left to sleep (will happen if it was interrupted by a signal)

Don't try to use this function to ensure that one thing happens before another

- `pause` puts calling function to sleep until a signal is received by the process.

```
int pause(void);
```

Don't use this function in a real program; use `sigsuspend`

`execve` Function

- Loads and runs a new program in the context of the current process.

argument list

executable object file

environment variable list

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

- `execve` returns to calling program only if there's an error.
  - called once, never returns
- `argv` and `envp` each point to a NULL-terminated array of pointers to strings.
  - by convention, `argv[0]` = name of the executable object file
  - each environment variable string has form "`NAME=VALUE`"

Example: `argv` and `envp`

```
/* myecho.c */

#include "csapp.h"

int main(int argc, char* argv[], char* envp[]) {
  int i;

  printf("Command line arguments:\n");
  for(i = 0; i < argc; i++)
    printf("\t argv[%2d]: %s\n", i, argv[i]);

  printf("Environment variables:\n");
  for(i = 0; envp[i] != NULL; i++)
    printf("\t envp[%2d]: %s\n", i, envp[i]);

  exit(0);
}
```

(See text for functions that manipulate `envp`.)

```
[user@lab1-3 ~]$ ./a.out hi
Command line arguments:
    argv[ 0]: ./a.out
    argv[ 1]: hi
Environment variables:
    envp[ 0]: HOSTNAME=lab1-3
    envp[ 1]: MALLOC_CHECK_=1
    envp[ 2]: TERM=xterm-256color
    envp[ 3]: SHELL=/bin/bash
    envp[ 4]: HISTSIZE=1000
    ...
```

Programs vs. Processes

- *Program*—collection of code and data
- *Process*—a specific instance of a program in execution
- `fork` runs the same program in a new child process that is a duplicate of the parent process.
- `execve` loads and runs a new program in context of the current process and *does not create a new process*.
  - new program has same PID
  - inherits all of the file descriptors that were open at the time of the call to `execve`

execve, to perform a sequence of read/evaluate steps.

Shells

- Read step—read a command line from the user.
- Evaluate step—parse the command line and run programs on the behalf of the user.
- *Simple shell example*:

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

  while(1) {
    printf("> ");
    Fgets(cmdline, MAXLINE, stdin);
    if(feof(stdin))
      exit(0);

    eval(cmdline);
  }
}
```

```
int parseline(char* buf, char** argv);
int builtin_command(char** argv);

void eval(char *cmdline) {     /* evaluate a command line */
  char *argv[MAXARGS]; /* argv for execve() */
  char buf[MAXLINE];   /* holds modified command line */
  int bg;              /* should the job run in bg or fg? */
  pid_t pid;           /* process id */

  strcpy(buf, cmdline);
  bg = parseline(buf, argv);  /* true if last argv is & */
  if(argv[0] == NULL)  return;   /* ignore empty lines */

  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);
      }

    /* parent waits for foreground job to terminate */
    if(!bg) {
      int status;
      if(waitpid(pid, &status, 0) < 0)
        unix_error("waitfg: waitpid error");
    }
    else
      printf("%d %s", pid, cmdline);
  }
  return;
}          /* shell is flawed because children not reaped */
```

Signals

- *Signal*—a message that notifies a process that an event of some type has occurred in the system.
  - allows processes to interrupt other processes
- Transfer of a signal to a destination process:

1. Kernel *sends* a signal to a destination process by updating some state in the context of the destination process.
2. A destination process *receives* a signal when it is forced by the kernel to react (ignore signal, terminate, or catch signal) to the delivery of the signal.

- (See text for a list of Linux signals.)

Pending Signals

- *Pending signal*—sent but not yet received.
- At any point, there can be at most one pending signal of a particular type.
- If a process *p* has a pending signal of type *k*, any subsequent signals of type *k* sent to *p* are discarded.
- A process can selectively block receipt of certain signals (signal is delivered, but not received until unblocked).
- A pending signal is received at most once.
- Kernel keeps track of pending and blocked signals.

Process Groups

- Every process belongs to exactly one *process group*.
  - a process group is identified by a process group ID > 0
  - `pid_t getpgrp(void)` returns process group ID of current process
- By default, a child process belongs to the process group of its parent.
- `setpgid` changes the process group of `pid` to `pgid`.

```
pid_t setpgid(pid_t pid, pid_t pgid);
```

  - if `pid`=0, PID of current process is used
  - if `pgid`=0, PID of process specified by `pid` is used for group id
  - what does `setpgid(0, 0)` do?

Sending Signals

- `kill` sends signal number `sig` to other process(es).
  - `int kill(pid_t pid, int sig);`
    - if `pid` > 0, sends to process `pid`
    - if `pid` < 0, sends to every process in process group *abs*(`pid`)

```
#include "csapp.h"

int main() {
  pid_t pid;

  /* child sleeps until SIGKILL signal received
     then dies */
  if((pid = Fork()) == 0) {
    Pause();  /* wait for signal */
    printf("control never reaches here");
    exit(0);
  }

  /* parent sends SIGKILL signal to child */
  Kill(pid, SIGKILL);
  exit(0);
}
```

Receiving Signals

- When the kernel is ready to pass control to process *p*, it checks the set of pending, unblocked signals.
  - if the set is empty, continue with I_{next} in *p*
  - otherwise, choose some signal number *k* (usually the smallest) from the set and force *p* to receive the signal
- The process completes some *action* in response and then control passes to I_{next}.
- Each signal has a default action (see text). Process either terminates, terminates and dumps core, stops until restarted by `SIGCONT` signal, or ignores signal.

Modifying Default

- `signal` modifies the default action for a signal.
  - `handler_t* signal(int signum, handler_t* handler);`
    - `handler` is the address of a user-defined function
    - default actions of `SIGSTOP` and `SIGKILL` cannot be changed
    - generally, sigaction() should be used instead of signal()

```
#include "csapp.h"

void handler(int sig) { /* SIGINT handler */
  printf("Caught SIGINT\n");
  exit(0);
}

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

  pause();  /* Wait for ctrl-c from keyboard */

  exit(0);
}
```

Explicitly Blocking Signals

- `sigprocmask` explicitly blocks selected signals.

```
int sigprocmask(int how, sigset_t* set, sigset_t* oldset);
```

- The set of blocked signals is maintained as a bit vector `blocked`.

- Behavior depends on argument `how`.
  - `SIG_BLOCK`—adds signals in `set` to `blocked` (`blocked |= set`)
  - `SIG_UNBLOCK`—removes signals in `set` from `blocked` (`blocked &= ~set`)
  - `SIG_SETMASK`—`blocked = set`

```
void handler(int sig) {
  pid_t pid;
  while((pid = waitpid(-1, NULL, 0)) > 0) /* Reap a zombie child */
    deletejob(pid); /* Delete the child from the job list */
  if(errno != ECHILD)
    unix_error("waitpid error");
}

int main(int argc, char** argv) {
  int pid;
  sigset_t mask;

  Signal(SIGCHLD, handler);
  initjobs(); /* Initialize job list (to keep track of children) */

  while(1) {
    Sigemptyset(&mask);
    Sigaddset(&mask, SIGCHLD);
    Sigprocmask(SIG_BLOCK, &mask, NULL); /* Block SIGCHLD */

    /* Child process */
    if((pid = Fork()) == 0) {
      Sigprocmask(SIG_UNBLOCK, &mask, NULL); /* Unblock SIGCHLD */
      Execve("/bin/ls", argv, NULL);
    }

    /* Parent process */
    addjob(pid);  /* Add the child to the job list */
    Sigprocmask(SIG_UNBLOCK, &mask, NULL);  /* Unblock SIGCHLD */
  }
  exit(0);
}
```

Nonlocal Jumps

- Transfer control from one function to another currently executing function, without having to go through the normal call-and-return sequence.
- `setjmp` saves the current stack context in `env`.

```
int setjmp(jmp_buf env);
```

- `longjmp` restores the stack context from the `env` buffer and then triggers a return from the most recent `setjmp` call that initialized `env`.

```
int longjmp(jmp_buf env, int retval);
```

  - `setjmp` then returns with return value `retval`

Nonlocal Jumps

- `setjmp` is called once and returns multiple times.
  - once when it is first called and stack context is saved
  - once for each corresponding call to `longjmp`
- `longjmp` never returns.
- Nonlocal jumps permit
  - immediate return from a deeply-nested function call, usually as a result of detecting some error (return directly to an error handler, rather than unwinding the call stack)
  - branching out of a signal handler to a specific code location, rather than returning to the instruction that was interrupted at the arrival of the signal

```
jmp_buf buf;

int error1 = 0;
int error2 = 1;

void foo(void), bar(void);

int main() {
  int rc;

  rc = setjmp(buf);  /* returns 0 when called directly */
  if(rc == 0)        /*   returns !=0 when called indirectly */
    foo();
  else if(rc == 1)
    printf("Detected an error1 condition in foo\n");
  else if(rc == 2)
    printf("Detected an error2 condition in foo\n");
  else
    printf("Unknown error condition in foo\n");
  exit(0);
}

void foo(void) {    /* deeply nested function foo */
  if(error1)
    longjmp(buf, 1);
  bar();
}

void bar(void) {
  if(error2)
    longjmp(buf, 2);
}
```

```c
/* restart.c */

sigjmp_buf buf;

void handler(int sig) {
  siglongjmp(buf, 1);   /* version of longjmp that can be */
}                       /*   used by signal handlers */
                        /*   1 means to restore the signal mask */
int main() {
  Signal(SIGINT, handler);

  if(!sigsetjmp(buf, 1))  /* version of setjmp for sig handlers */
    printf("starting\n"); /*   1 means to save the signal mask */
  else
    printf("restarting\n");

  while(1) {
    Sleep(1);
    printf("processing...\n");
  }
  exit(0);
}
```

```
unix> ./restart
starting
processing...
processing...
restarting          user types ctrl-c
processing...
restarting          user types ctrl-c
processing...
```

Summary

- ECF occurs at all levels of a computer system.
- *Hardware level*: interrupt, trap, fault, and abort classes of exceptions.
- *OS level*: a process provides the illusion that a program has exclusive use of the processor and memory.
- *Application level*: apps can create and wait for child processes, run new programs, and catch signals from other processes.
  - C programs can use nonlocal jumps to bypass the normal call/return stack discipline and branch directly to a function.