wait and waitpid (11.2)

- Recall from a previous slide: **pid_t wait(int *status)**
- **wait()** can: (a) block; (b) return with status; (c) return with error
- If there is more than one child, **wait()** returns on termination of *any* children
- **waitpid** can be used to wait for a specific child pid
- **waitpid** also has an option to block or not to block

pid_t waitpid(pid, &status, option);

pid == -1	waits for any child
option == NOHANG	non-blocking
option == 0	blocking

waitpid(-1, &status, 0) equivalent to **wait(&status)**

S-134

example: wait.c

```
#include <sys/types.h>
#include <sys/wait.h>
void main( void )
{
   int status;
   if( fork() == 0 ) exit( 7 );         /* normal exit */
   wait( &status ); prExit( status );

   if( fork() == 0 ) abort();           /* generates SIGABRT */
   wait( &status ); prExit( status );

   if( fork() == 0 ) status /= 0;       /* generates SIGFPE */
   wait( &status ); prExit( status );
}
```

S-135

prExit.c

```
#include <sys/types.h>
#include <sys/wait.h>
void prExit( int status )
{
   if( WIFEXITED( status ) )
      printf( "normal termination, exit status = %d\n",
                                  WEXITSTATUS( status ));
   else if( WIFSIGNALED( status ) )
      printf( "abnormal termination, signal number = %d\n",
                                  WTERMSIG( status ));
   else if( WIFSTOPPED( status ) )
      printf( "child stopped, signal number = %d\n",
                                  WSTOPSIG( status ));
}
```

S-136

exec

- Six versions of exec:

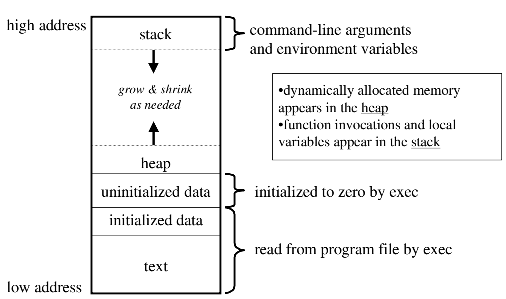
```
execl( char *pathname, char *arg0, ... , (char*) 0 );
execv( char *pathname, char *argv[] );

execle( char *pathname, char *arg0, ..., (char*) 0,
                                    char *envp[] );
execve( char *pathname, char *argv[],
                                    char *envp[] );

execlp( char *filename, char *arg0, ..., (char*) 0 );
execvp( char *filename, char *argv[] );
```

S-137

Memory Layout of a C program



S-138

Miscellaneous: permissions

- Read permissions for a directory and execute permissions for it are <u>not</u> the same:
  - **Read**: read directory, obtain a list of filenames
  - **Execute**: lets users pass through the directory when it is a component of a pathname being accessed
- Cannot create a new file in a directory unless user has write permissions and execute permission in that directory
- To delete an existing file, the user needs write and execute permissions in the directory containing the file, but does <u>not</u> need read or write permission for file itself (!!!)

S-139

Miscellaneous: buffering control

`int setbuffer(FILE *fp, char *buf, int size)`
- specifies that "**buf**" should be used instead of the default system-allocated buffer, and sets the buffer size to "**size**"
- if "**buf**" is **NULL**, i/o will be unbuffered
- used after stream is opened, but before it is read or written

`int setlinebuf( FILE *fp )`
- used to change **stdout** or **stderr** to line buffered
- can be called anytime

- A stream can be changed from unbuffered or line buffered to block buffered by using **freopen()**. A stream can be changed from block buffered or line buffered to unbuffered by using **freopen()** followed by **setbuf()** with a buffer argument of **NULL**.

S-140

Signals

S-141

Motivation for Signals (11.15)

- When a program forks into 2 or more processes, rarely do they execute independently of each other
- The processes usually require some form of synchronization, and this is typically handled using signals
- Data usually needs to be passed between processes also, and this is typically handled using pipes and sockets, which we'll discuss in detail in a week or two
- Signals are usually generated by
  - machine interrupts
  - the program itself, other programs, or the user (*e.g.* from the keyboard)

S-142

Introduction

- **<sys/signal.h>** lists the signal types on cdf. Table 11.5 and **signal(5)** give a list of some signal types and their default actions
- When a C program receives a signal, control is immediately passed to a function called a signal handler
- The signal handler function can execute some C statements and exit in three different ways:
  - return control to the place in the program which was executing when the signal occurred
  - return control to some other point in the program
  - terminate the program by calling the **exit** (or **_exit)** function

S-143

sigset()

- A default action is provided for each kind of signal, such as terminate, stop, or ignore
- For nearly all signal types, the default action can be changed using the **signal()** function. The exceptions are **SIGKILL** and **SIGSTOP**
- Usage: **signal(int sig, void (*disp)(int))**
- For each process, UNIX maintains a table of actions that should be performed for each kind of signal. The **signal()** function changes the table entry for the signal named as the first argument to the value provided as the second argument
- The second argument can be **SIG_IGN** (ignore the signal), **SIG_DFL** (perform default action), or a pointer to a signal handler function

S-144

sigset() example

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
int i = 0;
void quit( int code ) {
   fprintf( stderr, "\nInterrupt (code=%d, i=%d)\n", code, i );
   exit( 123 );
}
void main( void ) {
   if (signal( SIGINT , quit  ) == SIG_IGN) exit( 1 );
   if (signal( SIGTERM, quit  ) == SIG_IGN) exit( 2 );
   if (signal( SIGQUIT, quit  ) == SIG_IGN) exit( 3 );
   for(;;)
      if( i++ % 5000000 == 0 ) putc( '.', stderr );
}
```

S-145

Checking the return value

- The data type that **`signal()`** returns is:
  *pointer to function with **int** argument returning **void***
- So, the variable used to hold the result of a call to signal should be declared as follows:
  `void (*signal_result)(int);`
- It is possible for a child process to accept signals that are being ignored by the parent, which more than likely is undesirable
- Thus, the normal method of installing a new signal handler is:

```
oldhandler = sigset( SIGHUP, SIG_IGN );
if( oldhandler != SIG_IGN )
   sigset( SIGHUP, newhandler );
```

S-146

Signalling between processes

- One process can send a signal to another process using the misleadingly named function call
  `kill( int pid, int sig )`
- This call sends the signal "**`sig`**" to the process "**`pid`**"
- Signalling between processes can be used for many purposes:
  - kill errant processes
  - temporarily suspend execution of a process
  - make processes aware of the passage of time
  - synchronize the actions of processes

S-147

Timer signals

- Three interval timers are maintained for each process:
  - **`SIGALRM`** (real-time alarm, like a stopwatch)
  - **`SIGVTALRM`** (virtual-time alarm, measuring CPU time)
  - **`SIGPROF`** (used for profilers, which we'll cover later)
- Useful functions to set and get timer info are:
  - **`setitimer(), getitimer()`**
  - **`alarm()`** (simpler version: only sets **`SIGALRM`**)
  - **`pause()`** (suspend until next signal arrives)
  - **`sleep()`** (caused calling process to suspend)
  - **`usleep()`** (like **`sleep()`**, but with finer granularity)

Note: **`sleep()`** and **`usleep()`** are *interruptible* by other signals

S-148