PROCESSES AND SIGNALS

The wait Function The wait function is a simpler version of waitpid:

#include <sys/types.h> #include <sys/wait.h>

pid\_t wait(int \*status);

Returns: PID of child if OK or−1on error

Calling wait(&status) is equivalent to calling waitpid(-1, &status, 0).

The sleep function suspends a process for a speciﬁed period of time.

#include <unistd.h>

unsigned int sleep(unsigned int secs);

Returns: seconds left to sleep

Another function that we will ﬁnd useful is the pause function, which puts the calling function to sleep until a signal is received by the process.

#include <unistd.h>

int pause(void);

Always returns−1

The execve function loads and runs a new program in the context of the current process.

#include <unistd.h>

int execve(const char \*filename, const char \*argv[], const char \*envp[]); Does not return if OK, returns−1on error

The execve function loads and runs the executable object ﬁle filename with the argument list argv and the environment variable list envp. Execve returns to the calling program only if there is an error such as not being able to ﬁnd filename. So unlike fork, which is called once but returns twice, execve is called once and never returns.

#include <stdlib.h>

char \*getenv(const char \*name); Returns: ptr to name if exists, NULL if no match

The getenv function searches the environment array for a string “name=value”. If found, it returns a pointer to value, otherwise it returns NULL.

#include <stdlib.h>

int setenv(const char \*name, const char \*newvalue, int overwrite); Returns: 0 on success,−1on error

void unsetenv(const char \*name);

Returns: nothing

Iftheenvironmentarraycontainsastringoftheform“name=oldvalue”,then unsetenv deletes it and setenv replaces oldvalue with newvalue, but only if overwriteisnonzero.Ifnamedoesnotexist,thensetenvadds“name=newvalue” to the array.

#include <unistd.h>

pid\_t getpgrp(void);

Returns: process group ID of calling process

By default, a child process belongs to the same process group as its parent. A process can change the process group of itself or another process by using the setpgid function:

#include <unistd.h>

int setpgid(pid\_t pid, pid\_t pgid);

Returns: 0 on success,−1on error

The setpgid function changes the process group of process pid to pgid. Ifpid is zero,thePIDofthecurrentprocessisused.Ifpgidiszero,thePIDoftheprocess speciﬁedby pid isusedfortheprocessgroupID.Forexample, ifprocess15213is the calling process, then

setpgid(0, 0);

creates a new process group whose process group ID is 15213, and adds process 15213 to this new group

#include <sys/types.h> #include <signal.h>

int kill(pid\_t pid, int sig);

Returns: 0 if OK,−1on error

If pid is greater than zero, then the kill function sends signal number sig to process pid. If pid is less than zero, then kill sends signal sig to every process inprocessgroupabs(pid)

#include <unistd.h>

unsigned int alarm(unsigned int secs); Returns: remaining secs of previous alarm, or 0 if no previous alarm

The alarm function arranges for the kernel to send a SIGALRM signal to the calling process in secs seconds. If secs is zero, then no new alarm Is scheduled. In any event, the call to alarm cancels any pending alarms, and returns the number of seconds remaining until any pending alarm was due to be delivered (had not this call to alarm canceled it), or 0 if there were no pending alarms

#include <signal.h> typedef void (\*sighandler\_t)(int);

sighandler\_t signal(int signum, sighandler\_t handler); Returns: ptr to previous handler if OK, SIG\_ERR on error (does not set errno)

The signal functioncanchangetheactionassociatedwithasignal signum in one of three ways:

. If handler is SIG\_IGN, then signals of type signum are ignored. . If handler is SIG\_DFL, then the action for signals of type signum reverts to the default action. . Otherwise, handler is the address of a user-deﬁned function, called a signal handler, that will be called whenever the process receives a signal of type signum. Changing the default action by passing the address of a handler to the signal function is known as installing the handler. The invocation of the handler is called catching the signal. The execution of the handler is referred to as handling the signal.

#include <setjmp.h>

int setjmp(jmp\_buf env); int sigsetjmp(sigjmp\_buf env, int savesigs); Returns: 0 from setjmp, nonzero from longjmps

The setjmp function saves the current calling environment in the env buffer, for later use by longjmp, and returns a 0. The calling environment includes the program counter, stack pointer, and general purpose registers.

#include <setjmp.h>

void longjmp(jmp\_buf env, int retval); void siglongjmp(sigjmp\_buf env, int retval);

Never returns

The longjmp function restores the calling environment from the env buffer and then triggers a return from the most recent setjmp call that initialized env. The setjmp then returns with the nonzero return value retval.

THREADS

#include <pthread.h> typedef void \*(func)(void \*);

int pthread\_create(pthread\_t \*tid, pthread\_attr\_t \*attr, func \*f, void \*arg); Returns: 0 if OK, nonzero on error

The pthread\_create function creates a new thread and runs the thread routine f in the context of the new thread and with an input argument of arg

When pthread\_create returns, argument tid contains the ID of the newly created thread. The new thread can determine its own thread ID by calling the pthread\_self function.

#include <pthread.h>

pthread\_t pthread\_self(void);

Returns: thread ID of caller

Threadswaitforotherthreadstoterminatebycallingthepthread\_joinfunction.

#include <pthread.h>

int pthread\_join(pthread\_t tid, void \*\*thread\_return); Returns: 0 if OK, nonzero on error

Networking

A client establishes a connection with a server by calling the connect function.

#include <sys/socket.h>

int connect(int sockfd, struct sockaddr \*serv\_addr, int addrlen); Returns: 0 if OK,−1on error

The connect function attempts to establish an Internet connection with the server at socket address serv\_addr, where addrlen is sizeof(sockaddr\_in). The connectfunctionblocksuntileithertheconnectionissuccessfullyestablished or an error occurs. If successful, the sockfd descriptor is now ready for reading and writing, and the resulting connection is characterized by the socket pair

#include "csapp.h"

int open\_clientfd(char \*hostname, int port); Returns: descriptor if OK,−1on Unix error,−2 on DNS error

The open\_clientfd function establishes a connection with a server running on host hostname and listening for connection requests on the well-known port port.Itreturnsanopensocketdescriptorthatisreadyforinputandoutputusing the Unix I/O functions

#include <sys/socket.h>

int bind(int sockfd, struct sockaddr \*my\_addr, int addrlen); Returns: 0 if OK,−1on error

The bind function tells the kernel to associate the server’s socket address in my\_addr with the socket descriptor sockfd. Theaddrlen argument is sizeof(sockaddr\_in).

#include <sys/socket.h>

int listen(int sockfd, int backlog);

Returns: 0 if OK,−1on error

The listen function converts sockfd from an active socket to a listening socketthatcanacceptconnectionrequestsfromclients.Thebacklogargumentisa hint about the number of outstanding connection requests that the kernel should queue up before it starts to refuse requests

#include "csapp.h"

int open\_listenfd(int port);

Returns: descriptor if OK,−1on Unix error

The open\_listenfd function opens and returns a listening descriptor that is ready to receive connection requests on the well-known port port. Figure 11.17 showsthecodeforopen\_listenfd.Afterwecreatethelistenfdsocketdescriptor, we use the setsockopt function (not described here) to conﬁgure the server so that it can be terminated and restarted immediately