EE 367L Lab Client-Server

Due date: See laulima

The objective of this assignment is to gain experience with

- Client server paradigm
- Processes
- System calls (calls to functions in the operating system, which is assumed to be Linux)
- How data is transported over the Internet
- Communication between processes
    - pipes: communication between processes in a single machine
    - sockets: communication between processes over the Internet

The assignment is to write a simple client server system.

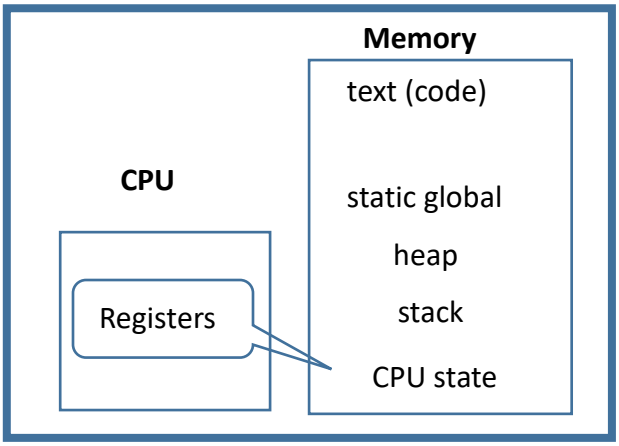
There are two parts to this lab:

- Part 1, Processes – Reading and a few exercises
    - What is a process
    - Managing processes
    - Launching programs from a process
    - Communicating processes: pipes
    - I/O redirection: dup2
    - More about processes
- Part 2, Internet – Reading and a few exercises (this will be in another document)
    - Brief overview of the Internet
    - Sockets
    - Explanation of a simple client-server example

1 Processes

What is a Process

Figure 1 shows a simple computer with a CPU and memory. A *process* is a running program, which includes the executable program (or machine program), the CPU and the memory to store data. The "state" of the computer are the values stored in the variables in memory (static global variables, and variables in the heap and stack), and registers in the CPU.



Example
Small microcontrollers
signal processors,
simple embedded systems

Capable of running one process

Figure 1. Simple Computer.

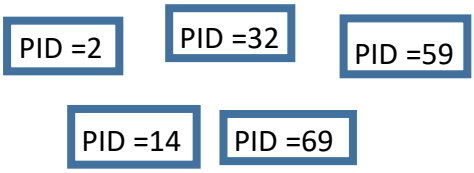
General purpose computers (e.g., laptops, smart phones, tablet, and servers) can run multiple processes. Each process has a portion of memory to store its text (code) and "state". As mentioned above, the "state" is the set of values in variables in memory and registers.

The computer can run these processes simultaneously, and to a large extend they run independently of each other. Thus, a process is basically a *virtual computer with a single CPU* running a single program. This is implemented by having the processes take turns running on the physical CPU. This is called *time sharing* the CPU. An active process is either physically running on a CPU, or is suspended. If it is suspended, it can resume running later.

Since a process has its own block of physical memory and consumes a certain amount of CPU processing bandwidth, more processes means more physical memory and CPU bandwidth are used.

An operating system (e.g., Linux) keeps track of its processes, each with its own ID number as shown in Figure 2.

General Purpose Computers run multiple processes
(e.g., Windows, Android, Apple OS, Linux)
Each process has its own unique Process ID (PID)



Processes run in "parallel" and independenty.
Processes are created/destroyed by the operating system

Figure 2. Multiple processes.

If you go to wiliki and type "ps -a", it will display the processes running in your account, e.g.,

PID	TTY	TIME	CMD
24538	pts/4	00:00:00	bash
24570	pts/4	00:00:00	ps

Each row is an active process:

- PID: Process ID. You can use this ID to manage the process. For example, you can kill (terminate) a process with PID 24538 by typing "kill 24538".
- CMD: Program name.

Processes are created over time. The first process has the PID equal to zero, and subsequent processes have higher PIDs. In the example, the processes have PIDs over 24000. This implies that a lot of processes have been created.

Managing Processes

Processes create other processes. The following are system calls that manage processes

- fork( ): This creates a process, which is referred to as a *child* process. The process that creates a child is called the *parent*.
    - When a process executes fork( ), it will create a child which is identical to itself. Both parent and child continue running; however, the returned value of fork( ) is different:
        - For the child, the returned value is zero.
        - For the parent, the returned value is the child's PID.
    - For example, suppose a parent process calls fork( ), which creates a child process with PID 423. Then the parent will receive a returned value of 423, while the child will receive a returned value of 0. In this way, the processes will know whether they are parent or child, e.g., they can check if the returned value is zero.
    - Note that a child process can create its own children, and becomes the parent of its children.

- exit( ): This exits (terminates) a process. exit(EXIT_SUCCESS) is used if the termination due to successful completion of the process, and exit(EXIT_FAILURE) is used if the termination is due to a failure. exit(0) and exit(1) are the older way of exit(EXIT_SUCCESS) and exit(EXIT_FAILURE), respectively.

- wait( ): This will cause a parent process to wait until a child (any child) terminates.

- waitpid(int pid): This is similar to wait( )
    - If pid > 0, then it waits for the child process with the PID pid to terminate.
    - If pid = -1, then it waits for any child process to terminate.
    - If pid = 0, then it waits for any child process in the same group as the parent process.

All of these calls require the unistd.h header, i.e.,

#include <unistd.h>

To find details about these functions, you can google them.

Read the following web site about fork( )
http://www.csl.mtu.edu/cs4411.ck/www/NOTES/process/fork/create.html
http://man7.org/linux/man-pages/man2/fork.2.html

Related system calls are wait( ) and waitpid( )
http://linux.about.com/od/commands/l/blcmdl2_wait.htm

Read the following about exit( )
http://linux.die.net/man/2/exit

A common application of fork( ) is

```
main( )
{
pid = fork( );
if (pid < 0) { /* error */
  indicate an error occured
}
if (pid== 0) { /* child process */
   processing for the child
}
else { /* parent process */
   processing for the parent
}

}
```

The child doesn't do the same thing as the parent because

- the child will continue running in "if (pid == 0)..."
- the parent will continue running in "else..."

Figure 3 shows what happens when main( ) is executed. Note that the call to fork() will cause the main process to be copied, where the copy is the child process. The parent and child processes are identical except that their pid values are different. The parent's pid will have the PID of the child, while the child's pid will be zero. Note that the program is designed so that the two processes will take different computation paths.

In theory, the two processes will run in parallel. In practice, processes can run on a single CPU, i.e., they share the CPU. The CPU will be switched between running the parent, then child, back again, and so forth. This is known as *time sharing*. The operating system, such as Linux or Android, will determine the switching. This is known as *CPU scheduling*, which is covered in EE 468 Operating Systems. In the simplest situations, the user has no control over the time sharing. Therefore, the operating system can dedicate more CPU bandwidth to one process over another which may be unwanted by the user. Though there are mechanisms that the user can apply to control the time sharing, it is beyond the scope of this lab.

Let us digress a bit to explain why processes are important. Processes allow a software designer the ability to develop large and complicated software systems. Typically, a program will have many tasks to accomplish, and these tasks are done repeatedly. The program can be designed so that each task is implemented by a process.

How processes are executed is managed by the operating system, e.g., through CPU scheduling. Typically, the operating system will manage the processes to make the most efficient use of the computer resources, such as the CPU and memory.

Thus, when developing a program, we can separate two design issues: (*i*) how to organize the program's functions to accomplish all its tasks and (*ii*) how to run the program efficiently. This separation greatly simplifies design.

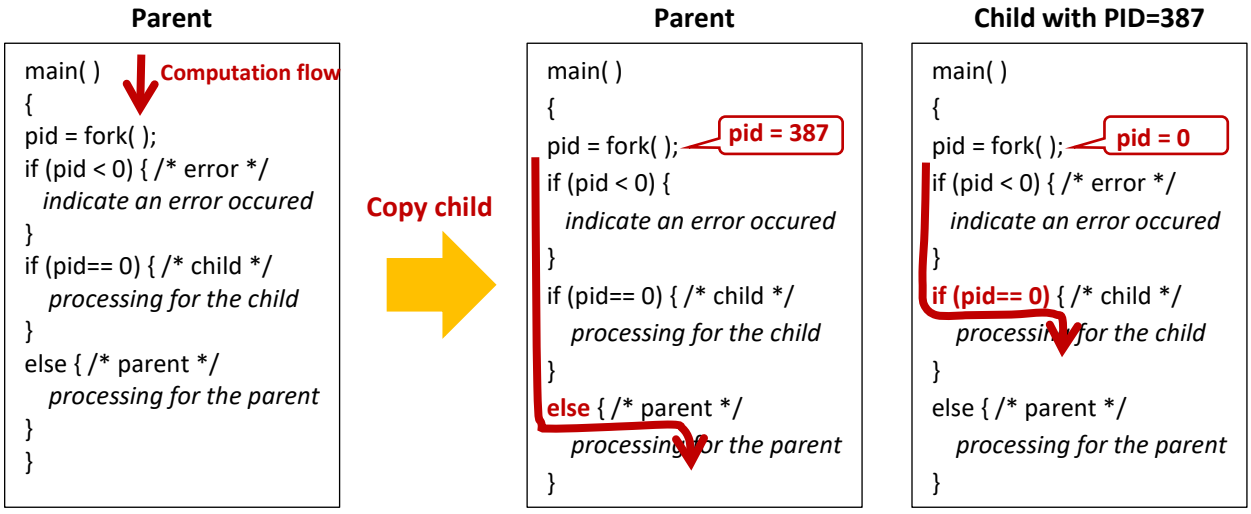


Figure 3. fork( ) operation.

The following is a more specific example application of fork( ). Consider the following for-loop, where slowTask() takes 5 seconds, and fastTask() takes 1 second.

```
for (int i=0; i<4; i++){
  slowTask( );
  fastTask( );
}
```

Each pass through this loop will takes 6 seconds, and the overall time of the loop is 24 seconds. To speed up the program, we can do the following:

```
for (int i=0; i<4; i++){
  if (fork( ) == 0) {  /* Child */
     slowTask( );
     exit(EXIT_SUCCESS);   /* Terminate child */
  }
  else { /* Parent */
     fastTask( );
  }
}
```

Then the parent will produce four children. Each child will take 5 seconds, and can run in parallel, if there is sufficient hardware. Then all the children will terminate in 5 seconds. The parent will go through the for-loop

4 times, and each pass will take 1 second. Thus, the parent will terminate n 4 seconds. Then the parent and its children will run in parallel, with the parent finishing in 4 seconds, and all the children terminating in 5 seconds. This is considerably faster than the 24 seconds of the previous example.

Suppose for each pass through the loop, we want slowTask() and fastTask() to run concurrently, but they should both be completed before the next pass. Then the child and parent must be synchronized. The following will have the parent wait for the slower child.

```
for (int i=0; i<4; i++) {
  if (fork( ) == 0) { /* Child */
    slowTask( );
    exit(EXIT_SUCCESS);
  }
  else { /* Parent */
    fastTask( );
    wait( ); /* wait for child to terminate */
  }
}
```

Here, the parent calls "wait( )", which causes the parent to wait until a child process terminates. The function wait( ) is called *blocking* because it suspends the parent process until an action occurs; in this case, the action is that a child process terminates. Note that you can use variations of wait( ) such as waitpid( ).

The following is another example, where 100 children are run concurrently per pass through the loop:

```
for (int i=0; i<4; i++) {
    for (int k=0; k<100; k++) {
        if (fork( ) == 0) {   /* Child */
            slowTask( );
            exit(EXIT_SUCCESS);
        }
    }
    fastTask( );   /* Task of the parent */
    for (int k=0; k<100; k++) {  /* Wait until 100 child processes terminate */
        wait(NULL );   /* Wait until a process terminates if there are active processes; otherwise, don't wait */
    }
}
```

The parent has a for-loop that loops four times. During each pass, the parent creates 100 children. Then it waits for all the processes to be completed. Assuming there are enough computing resources, the 100 children and the parent can run in parallel and take 5 seconds. The total time of the for-loop is 20 seconds.

Launching Programs From A Process

Find the following program aloha.c attached to this document:

aloha.c:

```
void main( )
{
printf("Aloha world!\n");
}
```

Compile it and name the executable "aloha". The following program launch.c shows how you can run "aloha" from another program. You can find launch.c attached to this document. Compile it and run it.

launch.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>   /* Necessary for 'execlp' */

void main()
{
printf("Start of program 'launch'\n");
execlp("./aloha","./aloha", (char *) NULL);
printf("End of program 'launch'\n");
}
```

The output will be

```
Start of program 'launch'
Aloha world!
```

Here's what happened: you started the process for 'launch', which displayed "Start of program 'launch'". Then it executed the function 'execlp'. This is a system call which replaces the process for 'launch' with a process that runs 'aloha'. At this point, the 'launch' process terminates and doesn't exist anymore. The process for 'aloha' continues running in its place.

The system call execlp() works as follows. First, execlp can have a variable number of inputs. For example, it can be called by

```
execlp( input 0,  input 1, input 2, input 3, ..., input 20, (char *) NULL);
```

or

```
execlp( input 0, input 1, input 2, input 3, (char *) NULL);
```

Since the number of inputs is variable, the last input is always the NULL value, more specifically (char *) NULL. In this way, NULL indicates the end of the input values.

Input 0 is a 'path' to the program. From our earlier example, './aloha' is the path to the program 'aloha'.

The rest of the inputs basically reflect how you would start program 'aloha' from the command line. In particular, we would type:

[sasaki@wiliki EE367/Spring19/Code]$ ./aloha

Therefore, input 1 will be "./aloha". Then input 2 will have the terminating value (char *) NULL.

You can find another example program add.c that's attached to this document. Here's the code:

add.c:

```
/*  Source code for add.c */
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
if (argc != 3) {
        printf("Usage:  ./add  <start value>  <end value>\n");
        return;
}
char *ptr;
int start = strtol(argv[1],&ptr,10);  /* Converts character string to long integer */
int end = strtol(arg[2],&ptr,10);
int sum = 0;
for (int i=start; i<=end; i++) {
        sum += i;
}
printf("Sum of %d to %d = %d\n"), start, end, sum);
}
```

Compile it and name the executable 'add'. Now type

./add 4 10

It adds the integers 4, 5, ..., 10 and prints

Sum of 4 to 10 = 49

Here is an example program that calls ‘add 4 10’

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main()
{
printf(“About to launch ‘add’\n”);
execlp(“./add”,”./add”, “4”, “10”, (char *) NULL);
printf(“The launch failed\n”);
}
```

Here’s another example which calls the Linux command: ls –l

```
void main()
{
execlp(“/bin/ls”, “ls”, “-l”, (char *) NULL);
}
```

Yet another example: cat <filename>

```
void main()
{
execlp(‘/bin/cat”, “cat”, “filename”, (char *) NULL);
}
```

There are other variations of execlp which you can find by searching the Internet. Here are a couple of links:
http://linux.die.net/man/3/exec
https://www.mkssoftware.com/docs/man3/execl.3.asp

An example variation is execvp. Attached to this document is a program run367.c which will run an executable program. For example,

run367 ./add 4 10

and

run367 ./aloha

will operate the same as if the user were to type “./add 4 10” or “./aloha” on the command line.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

void main(int argc, char * argv[])
{
if (argc < 2) {
        printf("Usage:  ./run367  <program>  <program input parameters if there are any>\n");
        return;
}
printf("About to launch '%s'\n", argv[1]);
char ** a = (char **) malloc(argc*sizeof(char*));
for (int k=0; k<argc-1; k++) {
        a[k] = argv[k+1];
}
a[argc-1] = (char *) NULL;
execvp(a[0],a);
printf("Launch failed\n");
}
```

Note that a[] is an array of char string pointers, where

- a[0] = argv[1]
- a[1] = argv[2]
- etc

and the final a[] value = (char *) NULL indicating the end of the input parameters. Note that argv[1] (= a[0]) is the program to launch, while a[1], a[2],.... are the input parameters.

Note that execlp has variable inputs but the variability is only before compile time. After compiling (i.e., during run time), the number of inputs is fixed. On the other hand, execvp allows variable number of inputs at run time.

In the examples given so far, the main program becomes replaced. Suppose we want the main program to continue after launching another program. Attached to this document is an example program multiexec.c which launches the program "add" four times as child processes. Basically multiexec.c will launch

```
./add  0  9
./add  1  10
./add  2  11
./add  3  12
```

multiexec.c:

```
#include <stdio.h>
#include <stdlib.h>

void error(char *s);

void main()
{
int pid;
for (int k=0; k<4; k++) {
        if ((pid=fork())<0) {
                printf("Could not fork a child\n");
                exit(EXIT_SUCCESS);
        }
        else if (pid == 0) {  /* Child */
                int length = snprintf(NULL, 0, "%d", k);  /* Compute char string length of representing int k */
                char* start = malloc(length + 1);     /* Allocate string including terminating NULL char */
                snprintf(start, length+1, "%d", k);   /* Convert k to string */
                length = snprintf(NULL, 0, "%d", k+9);
                char* end = malloc(length+1);
                snprintf(end, length+1, "%d", k+9);
                execlp("./add","./add", start, end, (char *) NULL);
                free(start);  /* You won't get this far unless execlp fails */
                free(end);
                error("Could not exec 'add' ");
        }
}
for (int k=0; k<4; k++) {
        wait(NULL);
}
printf("Parent: all children have terminated\n");
}

void error(char *s)
{
  perror(s); /* Enter a message on the standard error which usually some well known file */
  exit(EXIT_FAILURE);
}
```

Communicating Processes: Pipes

Previously, we showed how to create multiple processes that worked independently. However, there are many applications where we want processes to work together to solve a problem. Let us go through an example.

Suppose our task is to add the numbers 0 through 99. To take advantage of our multi-core processors, we will create four children, each summing a fourth of the numbers. In particular, the first child will sum 0 through 24, the second child will sum 25 through 49, and so forth. The results will be put in global variable "sum", and then displayed.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int sum;

void main()
{
sum = 0;
for (int k=0; k<4; k++) {
        int pid = fork();
        if (pid == 0) { /* Child */
                int s = 0;
                for (int i=0; i<25; i++) {
                        s += 25*k+i;
                }
                sum += s;
                exit(EXIT_SUCCESS);  /* Child terminates */
        }
}
for (int k=0; k<4; k++) {  /* Parent waits for all children to terminate */
        wait(NULL);
}
printf("Sum of 0 through 99 = %d\n",sum);
}
```

Unfortunately, this won't work because each child has its own global variable "sum". The parent's global variable "sum" remains 0 because it is unaffected by the children, who only update their own variable "sum". To operate correctly, the partial sums of the children must be transferred, or communicated, to the parent. Then the parent will add them to get the total sum.

A way to do this is with "pipes". A pipe is a memory buffer in the computer that multiple processes can access. Data is entered and retrieved from the pipe by using write( ) and read( ) system calls, respectively.

The following will create a pipe. Before you do this, you must create an int variable array to store the "file descriptor" of the pipe, which is the ID of the pipe.

```
int fd[2];   /* You must create a 2-dimension int array of size 2 */
```

The following creates the pipe and store's the pipe's file descriptor values in the array fd[].

```
pipe(fd);
```

Now

- fd[0] is the file descriptor of the read-end of the pipe – the output of the pipe
- fd[1] is the file descriptor of the write-end of the pipe – the input of the pipe

You can close the read-end using "close(fd[0])", and close the write-end using "close(fd[1])".

In many applications, a process will only use the write-end while another process will only use the read-end. These processes will close the end that it won't use.

The next page has an example which uses pipes:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

void int2bytes(int s, char result[]) /* Loads int ‘s’ into buffer result[] */
{
int val = s;
for (int k=0; k<4; k++) {
        result[k] = (char) (val & 255);  /* Mask everything except last byte */
        val = val >> 8;  /* Shift last byte out */
}
}

int bytes2int (char result[])  /* Returns the bytes in result[] as an int */
{
int val = 0;
for (int k=3; k>= 0; k--) {
        val = (val<<8) | (((int) result[k] &255);
}
return val;
}

void main(void)
{
int fd[4][2];
for (int k=0; k<4; k++) {
        pipe(fd[k]);  /* Create pipe for child k */
        int pid = fork();
        if (pid==0) {  /* Child */
                int s = 0;
                for (int i=0; i<25; i++) {
                        s +- 25*k + i;
                }
                char result[4];
                int2bytes(s,result);  /* Load the sum ‘s’ in the buffer called ‘result’ */
                close(fd[k][0]);  /* Close the output of the pipe, used for reading */
                write(fd[k][1], result, 4);  /* Write the result into the pipe */
                exit(EXIT_SUCCESS);  /* Child terminates */
        }
        close(fd[k][1]);  /* Parent closes the input to the pipe, used for writing */
}
```

```
/* Continued on the next page */
for (int k=0; k<4; k++) {  /* Parent waits for children to finish */
        wait(NULL);
}
int sum=0;
for (int k=0; k<4; k++) {
        char buffer[4];
        read(fd[k][0], buffer, 4);
        sum += bytes2int(buffer);
}
printf("Sum of 0..99 = %d\n",sum);
}
```

You can find other examples of using the pipe and fork() on the internet by doing searches. The following comes from http://tldp.org/LDP/lpg/node11.html, which shows an example of a child process sending a character string to the parent process over a pipe.

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>

int main(void)
{
        int     fd[2], nbytes;
        pid_t   childpid;
        char    string[] = "Hello, world!\n";
        char    readbuffer[80];

        pipe(fd);

        if((childpid = fork()) == -1)
        {
                perror("fork");
                exit(1);
        }

        if(childpid == 0)
        {
                close(fd[0]); /* Child process closes up output side of pipe */

                /* Send "string" through the input side of pipe */
                write(fd[1], string, (strlen(string)+1));
                exit(0);
        }
        else
        {
                close(fd[1]); /* Parent process closes up input side of pipe */

                /* Read in a string from the pipe */
                nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
                printf("Received string: %s", readbuffer);
        }
        return(0);
}
```

Note that both “write()” and “read()” have three input parameters:

- File descriptor
- A reference to a byte array, such as a character array
- An integer value, which is the maximum length of bytes to be sent or received.

The returned valued is the actual number of bytes that were sent or received. This value is important since “write” or “read” may not transfer the maximum number of bytes. Also, note that the pipe was created before the fork( ) so that both parent and child have access to it.

Note that when a read() or write() operation occurs on the pipe, ideally it does it in a single atomic action. The pipe has a capacity that depends on the operating system, but typical values are 16KB and 64KB.

The pipe can be *blocking* or *nonblocking*. Blocking means that if the pipe is empty, a read() operation is blocked (i.e., the process is suspended) until the pipe is not empty. Nonblocking means that a read() operation operation does not block the process. So if the pipe is empty, the read() will return a value of 0 (indicating no bytes are read) or -1 (indicating read error).

See the attached code pipeblock.c. It shows how to make a pipe nonblocking by using fcntrl( ).

For this assignment, we will use the default blocking configuration.

What is a file descriptor? Recall that file descriptors are used to reference files, e.g., “fd = fopen(“filename”);”. They are used as references to other objects such as pipes. These references point to all the information needed to manage the object such as

- Methods (the following may be stored as pointers to functions that do the operations)
    - open( );
    - close( );
    - create( );
    - write( );
    - read( );
- Metadata
    - Pointer to where the data is physically stored
    - Status of the object, e.g., Read-only, Write-only, Read-write

Here is a tutorial on file descriptors: https://www.geeksforgeeks.org/input-output-system-calls-c-create-open-close-read-write/

Redirecting I/O: dup2

Let's reexamine the program that adds the numbers 0 through 99, by creating four children, each summing a fourth of the numbers. In particular, the first child will sum 0 through 24, the second child will sum 25 through 49, and so forth.

Attached to this document is the program add2.c. Compile it and rename the executable 'add2'. The source code is shown below.

add2.c:

```
#include <stdlib.h>
#include <stdio.h>

void main(int argc, char *argv[])
{
if (argc != 3) {
        printf("0\n");
        return;
}
char *ptr;
int start = strtol(argv[1],&ptr,10);  /* Converts character string to long integer */
int end = strtol(arg[2],&ptr,10);
int sum = 0;
for (int i=start; i<=end; i++) {
        sum += i;
}
printf("%d\n"), sum);
return;
}
```

By typing:

```
./add2  1 3
```

will display

```
6
```

The program 'add2' takes its input from the command line from the keyboard, and outputs its result on the terminal. By default the command line from the keyboard is the *standard input* (stdin) and the terminal is the *standard output* (stdout). The file descriptors for the standard input and output is 0 (STDIN_FILENO) and 1 (STDOUT_FILENO), respectively. There is also a standard error (stderr) which has file descriptor 2 (STDERR_FILENO). The standard error refers to an error log located somewhere, and don't worry about this for now. To use the macros STDIN_FILENO, STDOUT_FILENO, and STDERR_FILENO you need unistd.h.

We want to call add2 to do our partial sums by creating child processes, which run ‘add2’. We want the output of add2 to go directly to pipes connected to the parent rather than the terminal. So we will change the file descriptor for STDOUT_FILENO to our pipe by using a system call dup2().

There is an attached program multipipe.c that does this. Compile and run it.

multipipe.c:

```
void main(void)
{
int pid;
int fd[4][2];
for (int k=0; k<4; k++) {
        pipe(fd[k]);  /* Create pipe for child k */
        if ((pid=fork())<0) {
                printf("Could not fork a child\n");
                exit(EXIT_FAILURE);
        }
        else if (pid == 0) {  /* Child */
                close(fd[k][0]);  /* Close the output of the pipe, used for reading */
                dup2(fd[k][1], STDOUT_FILENO);  /* Replace stdout with pipe */

                int length = snprintf(NULL, 0, "%d", k);  /* Compute char string length of representing int k */
                char* start = malloc(length + 1);     /* Allocate string including terminating NULL char */
                snprintf(start, length+1, "%d", k);   /* Convert k to string */
                length = snprintf(NULL, 0, "%d", k+9);
                char* end = malloc(length+1);
                snprintf(end, length+1, "%d", k+9);

                execlp("./add2","./add2", start, end, (char *) NULL);
                free(start);  /* You won't get this far unless execlp fails */
                free(end);
                error("Could not exec 'add2' ");  /* Defined earlier */
        }
        close(fd[k][1]);  /* Parent closes the input to the pipe, used for writing */
}
for (int k=0; k<4; k++) {  /* Wait for all processes to finish */
        wait(NULL);
}
int sum=0;
char buffer[100];
for (int k=0; k<4; k++) { /* Parent collects and sums the results from children */
        int j = read(fd[k][0], buffer, 100);
        if (j<100) { /* Received sum is not in error */
                buffer[j] = NULL;
                sum += (int) strtol(buffer, NULL, 10);
        }
}
printf("Parent: sum=%d\n", sum);
}
```

Note that dup2(fd[k][1], STDOUT_FILENO) copies the parameters of the file descriptor fd[k][1] to the parameters of the file descriptor STDOUT_FILENO, which is 1. So every access to file descriptor 1 goes to the pipe fd[k][1].

Attached is a program pipe.c that shows a parent creating a child, which runs the Linux program hexdump. The program hexdump inputs a file, and then displays each byte of the file as a hexadecimal digit. The parent sends a message to the child (hexdump), which sends its output back to the parent. (pipe.c was found at http://stackoverflow.com/questions/70842/execute-program-from-within-a-c-program)

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void error(char *s);
char *data = "Some input data\n";

main()
{
  int in[2], out[2], n, pid;
  char buf[255];

  if (pipe(in) < 0) error("pipe in");  /* Create pipe 'in' */
  if (pipe(out) < 0) error("pipe out"); /* Create pipe 'out' */
  if ((pid=fork()) == 0) { /* Child */
    close(0); /* Close stdin, stdout, stderr */
    close(1);
    close(2);
    dup2(in[0],0); /* Redirect stdin, stdout, and stderr to/from pipes */
    dup2(out[1],1);
    dup2(out[1],2);
    close(in[1]); /* Close unused ends of pipes */
    close(out[0]); /* Then when the other end is closed the processes get EOF */
    execl("/usr/bin/hexdump", "hexdump", "-C", (char *)NULL);
    error("Could not exec hexdump");
  }
  /*  Parent process */
  printf("Spawned 'hexdump -C' as a child process at pid %d\n", pid);
  close(in[0]); /* Close unused end of pipes */
  close(out[1]);
  printf("String sent to child: %s\n\n", data);
  write(in[1], data, strlen(data)); /* Parent sends string to hexdump */
  close(in[1]); /* This will for an EOF to be sent to child */
  n = read(out[0], buf, 250); /* Read back any output from hexdump */
  buf[n] = 0;
  printf("This was received by the child: %s",buf);
  exit(0);
}

void error(char *s)
{
  perror(s);
  exit(1);
}
```

More About Processes

- Processes take up memory space
    - If you have too many processes, you can get locked out of your account – it fills your memory allocation
- Zombie processes are ones that are improperly terminated
    - They run and take up space, and you may not know it
    - They survive even after you logout and login
- Terminating a process manually
    - kill <PID>
    - Ctrl-C if it's a foreground process
- Terminating a process in code
    - Use exit() at the end
- List all processes in your space
    - ps –a
    - You should do this periodically, so you don't accumulate zombie processes
- Foreground/background
    - A process can be run in foreground or background
    - Foreground process can interact with the user – you can kill it using ctrl-C
- Background processes do not interact with the user
    - You can kill it using kill <PID>
- To put a process running in foreground into background
    - Type ctrl-Z – this suspends the process
    - Type bg <PID> -- this puts the suspended foreground process in background – or just type bg
- To put a background process into foreground
    - Type fg <PID>
- To run a program a.out into background
    - Type ./a.out &
    - This is useful when you run your server process

Part 2 is another document.