An Introduction to Concurrency in Unix-based [GNU] C Through Annotated Examples

by
Henry M. Walker
Grinnell College, Grinnell, IA

Program 1: fork-1.c

This program creates a new process and prints some process id's.

```
/* This program creates a new process and allows both child and parent
   to report their idenfication numbers. */

#include <sys/types.h>       /* file of data types needed for many compilers */
#include <unistd.h>          /* needed for fork, getpid procedures */
#include <stdio.h>

int main (void)
{  pid_t pid;                /* variable to record process id of child */

   pid = fork();             /* create new process */
   if ( -1 == pid)           /* check for error in spawning child process */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)             /* check if this is the new child process */
     { /* processing for child */
       printf ("This output comes from the child process\n");
       printf ("Child report:  my pid = %d\n", getpid());
     }
  else
     { /* processing for parent */
       printf ("This output comes from the parent process.\n");
       printf ("Parent report:  my pid = %d   child's pid = %d\n",
                getpid(), pid);
     }

   exit (0);                 /* quit by reporting no error */
}
```

Annotations for `fork-1.c`*:*

- The commentary for this and subsequent programs assumes a basic knowledge of C's standard libraries, I/O, control structures, procedures, and parameters.
- Process control procedures typically use the `#include <unistd.h>` library. On many systems, this also requires the `#include <sys/types.h>` library.
- Each process has a process identification number, which has type `pid_t`.
- The `fork` procedure spawns a new process, copying exactly the elements of the current process. The new process is called a *child* process, and the previously existing process is called the *parent*.
  - § That is, the newly created process gets a copy of the parent's data speace, heap, and stack.
  - § Open file attributes are copied.
- Both the parent and child continue execution at the point following the call to `fork`. The behavior of the two processes is identical, except for the value returned by `fork`:
  - § `fork` returns the process id of the child to the parent, assuming the `fork` was successful. (If a new process cannot be created, `fork` returns –1 to the parent.)
  - § `fork` returns 0 to the child process.

- After the `fork` operation, the two processes execute independently.
- Since process operations can fail, sometimes leaving administrative and bookkeeping details behind, programs always should test such calls for errors and provide an appropriate and graceful exit if errors have occured. This is commonly done in two steps:
  - § `perror()` writes an error message on standard error, based on an internal error coding variable – `errno`.
  - § `exit(1);` causes the current process to terminate and returns an integer variable (`status`) to the operating system for further inspection. Usually, `exit(0);` indicates no error has occured, while a nonzero value (e.g., `exit(1);`) indicates some error has occurred.
- Following error checking, the two processes (parent and child) usually go their separate ways. This is accomplished by testing the value of the returned process id.
- Any process may obtain its own process id with the function `getpid()`. Similarly, a process may obtain the process id of its parent with `getppid()`.

Sample Run of `fork-1.c`*:*

```
babbage% gcc -o fork-1 fork-1.c && fork-1
This output comes from the child process
Child report:  my pid = 24420
This output comes from the parent process.
Parent report:  my pid = 24419   child's pid = 24420
```

This sample run (from the machine *babbage*) compiles and runs the program using the standard gcc compiler.

After starting one program, both processes are generated and report their identification numbers.

Program 2: fork-2.c

Program illustrating interprocess communication using a Unix pipe to read and write.

```
/* This program creates a new process and provides a simple pipe to
   allow the parent to communicate with the child
   Version 1 -- explicitly using pipe and byte-oriented read/write */

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

#define MAX 25            /* maximum number of letters communicated */

int main (void)
{  int fd[2];             /* provide file descriptor pointer array for pipe */
                          /* within pipe:
                                 fd[0] will be input end
                                 fd[1] will be output end */
   pid_t pid;

   char line[MAX];        /* character array (string) for reading */

   if (pipe (fd) < 0)     /* create pipe and check for an error */
     { perror("pipe error");
       exit (1);
     }

   if ((pid = fork()) < 0)  /* apply fork and check for error */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)
     { /* processing for child */
       printf ("The child process is active.\n");
       close (fd[1]);       /* close output end, leaving input open */
       read(fd[0], line, MAX);
       printf ("The string received is '%s'\n", line);
     }
   else
     { /* processing for parent */
       printf ("The parent process is active.\n");
       close (fd[0]);       /* close input end, leaving output open */
       write (fd[1], "Your parent is calling", 23);
                            /* print string and indicate byte length */
     }

   exit (0);                 /* quit by reporting no error */
}
```

Annotations for `fork-2.c`*:*

- Interprocess communication in Unix may be accomplished at a low level using pipes. A Unix pipe is a circular buffer maintained by the operating system. Typically, one or more processes write data into the buffer; another process reads the data from the pipe.
- While Unix handles the internals of pipe communication, programs communicate through pipes at a slightly more conceptual level. More specifically, the use of pipes involves four main steps
  1. An array of two integers is declared.
  2. The array is initialized by the `pipe` procedure. With this initialization, the first array element (subscript 0) provides an appropriate file descriptor for reading, while the second array element (subscript 1) provides an appropriate file descriptor for writing.
  3. After the `fork()` operation, both the reading and writing ends of the pipe are available to both processes. Traditionally, however, pipes are available for one-way communication only. Thus, the reading end of the pipe should be closed off in the writing process, and the writing end of the pipe should be closed off in the reading process.
  4. Data are moved byte-by-byte through a pipe, using `read` and `write` system calls.
- The `write` call requires three parameters, the output file number, a base address (e.g., a character string or an array of characters), and the number of characters to be written.
- The `read` call requires three parameters, the input file number, a base address (e.g., a character string), and a number of bytes. Reading retrieves the next number of byes – up to the size of the buffer.

Sample Run of `fork-2.c`*:*

```
babbage% gcc -o fork-2 fork-2.c && fork-2
The child process is active.
The parent process is active.
The string received is 'Your parent is calling'
babbage% fork-2
The parent process is active.
babbage% The child process is active.
The string received is 'Your parent is calling'
fork-2
The parent process is active.
babbage% The child process is active.
The string received is 'Your parent is calling'
```

When the program is compiled and run, the parent process writes data to the pipe and then terminates. The child process reads the string from the pipe and prints the result. The parent may start and finish either before or after the child, so the order of the output may differ. Further, once the parent process finishes, the shell prompt (e.g., `babbage%`) appears and `dtterm` is ready for the next command. If the child has not yet completed its work, then the child's output may appear after the shell prompt. The above output shows both of these cases on subsequent runs.

Program 3: fork-3.c

Program uses `write` and `read` for integer values, illustrates local and global variables for forked processes, and requires the parent wait for the child process to finish.

```
/* This program creates a new process and provides a simple pipe to
   allow the parent to communicate with the child
   Version 2 -- explicitly using pipe, read/write for integers, and wait */

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

int data_g = 4;           /* global data element */

int main (void)
{  int fd[2];             /* provide file descriptor pointer array for pipe */
                          /* within pipe:
                                fd[0] will be input end
                                fd[1] will be output end */
   pid_t pid;

   int value, value1;     /* integer values for writing and reading */
   int data_l= 5;         /* local data element */

   if (pipe (fd) < 0)     /* create pipe and check for an error */
     { perror("pipe error");
       exit (1);
     }

   if ((pid = fork()) < 0)  /* apply fork and check for error */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)
     { /* processing for child */
       printf ("The child process is active.\n");
       close (fd[1]);       /* close output end, leaving input open */
       read (fd[0], &value1, 4);  /* read integer as 4 bytes */
       printf ("The value read from the parent is %d\n", value1);
       printf ("Final data elements in child: global = %d   local = %d\n",
               data_g, data_l);
       printf ("Child finished\n");
     }
   else
     { /* processing for parent */
       printf ("The parent process continues.\n");
       data_g = 10;         /* change data elements */
       data_l = 12;
       value = 20;

       printf ("New data elements in parent: global = %d   local = %d\n",
               data_g, data_l);
       close (fd[0]);       /* close input end, leaving output open */
       write (fd[1], &value, 4); /* print value, which uses 4 bytes */
       waitpid (pid, NULL, 0);   /* suspend processing until child finish */
```

```
        printf ("Parent finished\n");
      }

    exit (0);
}
```

Annotations for `fork-3.c`*:*

- The `pipe` operation initializes system-level I/O. In particular, `write` and `read` require the address of data for their second parameter and a number of bytes to transfer as their third parameter.
  - § Unlike `printf`, both `write` and `read` require an address for the second parameter. Thus, the address operation & is used here.
  - § Since standard integers on MathLAN are 4 bytes long, the example shows the third parameter for both `write` and `read` as 4.
- As previously stated, the `fork` operation copies the entire process environment in creating the child process. Thus, after a child starts, the parent and the child have separate copies of all variables – either local or global. Changes in one copy by one process has no affect the other.
- The `wait()` system call suspends process activity until some child processes have completed.
- An alternative form of `wait(pid)` requires a process identification `pid` as parameter. In this case, `wait(pid)` suspends execution of the process until the process with the designated id is finished.

Sample Run of `fork-3.c`*:*

```
babbage% gcc -o fork-3 fork-3.c && fork-3
The child process is active.
The parent process continues.
New data elements in parent: global = 10   local = 12
The value read from the parent is 20
Final data elements in child: global = 4   local = 5
Child finished
Parent finished
```

While the message passing from parent to child guarantees that the values of data variables are changed in the parent before being printed by the child, the corresponding values in the child remain changed. This highlights that `fork` creates distinct data spaces for the two processes.

The `wait` within the parent process guarantees that the parent will finish after the child. Thus, control will not return to the shell until all work is done, and the resulting shell prompt will come after other output.

Program 4: fork-4.c

Program which spawns two processes and which allows the two childred to communicate through standard input and standard output.

```
/* This program creates a new process and provides a simple pipe to
   allow two children to communicate
   Version 3 -- using pipe for standard input and standard output */

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

#define MAX 4

int main (void)
{  int fd[2];             /* provide file descriptor pointer array for pipe */
                          /* within pipe:
                                 fd[0] will be input end
                                 fd[1] will be output end */
   char line[MAX];        /* line for reading */

   pid_t pid1, pid2;      /* process id's for each child */

   int value, value1;     /* integer values for writing and reading */

   if (pipe (fd) < 0)     /* create pipe and check for an error */
     { perror("pipe error");
       exit (1);
     }

   if ((pid1 = fork()) < 0)  /* apply fork and check for error */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid1)
     { /* processing for child */
       printf ("The first child process is active.\n");
       close (fd[1]);       /* close output end, leaving input open */
       /* set standard input to pipe */
       if (fd[0] !=  STDIN_FILENO)
         {
           if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
             { perror("dup2 error for standard input");
               exit(1);
             }
           close(fd[0]); /* not needed after dup2 */
         }

       gets (line);   /* read line from the pipe, now standard input */
       printf ("The string read from the parent is '%s'\n", line);

       printf ("First child finished\n");
     }
   else
     { /* processing for parent */
```

```
        printf ("The parent process continues.\n");

        /* spawn second process */
        if ((pid2 = fork()) < 0)  /* apply fork again for second child*/
          { perror ("error in fork");
            exit (1);
          }

        if (0 == pid2)
         { /* processing for child */
           printf ("The second child process is active.\n");
           close (fd[0]);       /* close input end, leaving output open */
           /* set standard output to pipe */
           if (fd[1] !=  STDOUT_FILENO)
             {
               if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                 { perror("dup2 error for standard output");
                   exit(1);
                 }
               close(fd[1]); /* not needed after dup2 */
             }

           printf ("Message from second child: hi\n");
                                /* print to the pipe, now standard output */
           /* note:  cannot print completion message to keyboard,
                     as stdout now changed to pipe! */

         }
        else
         { /* processing continues for parent */
           printf ("Parent waits for both children to finish.\n");
           waitpid (pid1, NULL, 0); /* wait for first child to finish */
           waitpid (pid2, NULL, 0); /* wait for second child to finish */
           printf ("Parent finished.\n");
         }
      }

    exit (0);
}
```

Annotations for `fork-4.c`*:*

- To spawn two children, the main program uses `fork` twice, following much the same approach discussed previously.
- Once a pipe is created and checked for error, the two children close the appropriate input or output end using `close (fd[--])`, as before.
- To set standard input to reference the pipe, one uses `dup2` to duplicate the pipe descriptor for input (`fd[0]`) onto standard input, `STDIN_FILENO)`. In most cases, `dup2` copies the file descriptor onto the second. The testing handles some special cases:
  - § If the two arguments are equal (e.g., if `fd[0] == STDIN_FILENO`, then duplicating the file descriptor would be unnecessary. Further, in this case, `dup2` would return a single file descriptor. Thus, closing one would close the other – an undesired result for this application.
  - § In normal operation, `dup2` returns the second file descriptor. Thus, an error must have resulted if `(dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)`.

- Setting standard output to the pipe similarly calls `dup2`, matching the pipe input `fd[1]` with `STDOUT_FILENO)`.
- Once standard input and output reference a pipe, all the familiar I/O operations apply.
  § `printf` and `scanf` perform formatted I/O.
  § `gets` allows the reading of a full line of input.
- By keeping track of each child process id, the parent process can wait for each child to finish by specify `waitpid()` with the appropriate process id's as parameters.

Sample Run of `fork-4.c`*:*

```
babbage% gcc -o fork-4 fork-4.c && fork-4
The parent process continues.
The second child process is active.
The first child process is active.
The string read from the parent is 'Message from second child: hi'
First child finished
Parent waits for both children to finish.
Parent finished.
```

Program 5: fork-5.c

Program sets pipe ends to standard I/O and calls `exec`s operations to run some other programs.

```
/* This program creates a new process and provides a simple pipe to
   allow the parent to communicate with the child
   Version 4 -- using exec's to run external programs*/

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

int main (void)
{  int fd[2];              /* provide file descriptor pointer array for pipe */
                           /* within pipe:
                                  fd[0] will be input end
                                  fd[1] will be output end */

   pid_t pid1, pid2;       /* process id's for each child */

   int value, value1;      /* integer values for writing and reading */

   if (pipe (fd) < 0)      /* create pipe and check for an error */
     { perror("pipe error");
       exit (1);
     }

   if ((pid1 = fork()) < 0)  /* apply fork and check for error */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid1)
     { /* processing for child */
       printf ("The first child process is active.\n");
       close (fd[1]);       /* close output end, leaving input open */
       /* set standard input to pipe */
       if (fd[0] !=  STDIN_FILENO)
         { if (dup2(fd[0], STDIN_FILENO) != STDIN_FILENO)
             { perror("dup2 error for standard input");
               exit(1);
             }
           close(fd[0]); /* not needed after dup2 */
         }

       execlp ("sort", "sort", "-n", "+5", (char *) 0);
       printf ("First child finished\n");
     }
   else
     { /* processing for parent */
       printf ("The parent process continues.\n");

       /* spawn second process */
       if ((pid2 = fork()) < 0)  /* apply fork again for second child*/
         { perror ("error in fork");
           exit (1);
         }

       if (0 == pid2)
        { /* processing for child */
          printf ("The second child process is active.\n");
          close (fd[0]);       /* close input end, leaving output open */
          /* set standard output to pipe */
          if (fd[1] !=  STDOUT_FILENO)
            { if (dup2(fd[1], STDOUT_FILENO) != STDOUT_FILENO)
                { perror("dup2 error for standard output");
                  exit(1);
                }
             close(fd[1]); /* not needed after dup2 */
            }

         execlp("cat", "cat", "/home/walker/151s/labs/ia-senate", (char *) 0);
                             /* print to the pipe, now standard output */
        }
       else
        { /* processing continues for parent */
          printf ("Parent closing its pipe ends: parent does not use pipe\n");
          close(fd[0]);
          close(fd[1]);
          printf ("Parent waits for both children to finish.\n");
          waitpid (pid1, NULL, 0); /* wait for first child to finish */
          waitpid (pid2, NULL, 0); /* wait for second child to finish */
          printf ("Parent finished.\n");
        }
     }

   exit (0);
}
```

Annotations for `fork-5.c`*:*

- This program spawns two child processes and maps the ends of a pipe to standard in/out, as in the previous example.
- Rather than writing separate application code for each child process, each child sets up standard I/O and then executes a separate, external program.
  § Child 1 executes the Unix-utility `sort`.
  § Child 2 executes the Unix-utility `cat`.
- Altogether, `cat` reads the file `"/home/walker/151s/labs/ia-senate"` which contains a list of members of the Iowa Senate for 1997-1998. These data are then sent through pipe as standard out. The result is the same as giving the command
  `cat "/home/walker/151s/labs/ia-senate"`
  in a Unix-shell window.
- Similarly, `sort` reads the data from standard input and orders the lines numerically according to the 6th column (skipping the first 5 columns).
- Coupling `cat` and `sort` through this pipe is equivalent to the Unix-shell command:
  `cat "/home/walker/151s/labs/ia-senate" | sort -n +5`
- C contains six versions of `exec` procedures to execute programs. Many details are available from system documentation by typing `man exec` at a Unix-shell prompt.

- The sample call `execlp ("sort", "sort", "-n", "+5", (char *) 0)` illustrates the major elements of the `execlp` call:
  - § `execlp` takes several parameters. The first two specify the name of the program to be called. The "p" in `execlp` specifies that the system will follow the normal search path to locate the program named.
  - § The "l" in `execlp` indicates that command-line arguments to the program will be specified as a list of string parameters. (Technically, the first parameter to `execlp` is the program name, and the command line arguments follow. However, since the first argument of any program call is the program's name itself, the second parameter to `execlp` always repeats the program name.) In this example, two additional parameters are given: `"-n"` indicates sorting is to consider the values as numbers, and `"+5"` indicates that ordering should be based on the sixth and later columns – skipping over the first 5 columns of the output.
  - § The final parameter to `execlp` is always the null character `(char *) 0`, which terminates the parameter list.
  - § A call to `execlp` does not return; the process terminates when the called program is done. Thus, the final statement for child 1, `printf ("First child finished\n")` is never executed.
- Sorting can only finish when `sort` knows it has read all data. This is accomplished when all processes close the pipe for writing. This is done explicitly at the start of child 1 (`close (fd[1])`) and implicitly by the end of `execlp("cat", ...)`. Since the parent process also could potentially write to this pipe, however, the parent also must close the same pipe for writing. Hence the final `close (fd[1])` for the parent process.
- As a dutiful parent, the parent process waits until all its children are done before terminating itself.

The raw data file, "/home/walker/151s/labs/ia-senate", is shown below:

```
                Members of the 1997-1998 Iowa Senate

Angelo          Jeff       44     Creston          IA 50801
Bartz           Merlin     10     Grafton          IA 50440
Behn            Jerry      40     Boone            IA 50036
Black           Dennis     29     Grinnell         IA 50112
Black           James       8     Algona           IA 50511-7067
Boettger        Nancy      41     Harlan           IA 51537
Borlaug         Allen      15     Protivin         IA 52163
Connolly        Mike       18     Dubuque          IA 52002
Dearden         Dick       35     Des Moines       IA 50317
Deluhery        Patrick    22     Davenport        IA 52804
Douglas         JoAnn      39     Adair            IA 50002
Drake           Richard    24     Muscatine        IA 52761
Dvorsky         Robert     25     Coralville       IA 52241
Fink            Bill       45     Carlisle         IA 50047
Flynn           Tom        17     Epworth          IA 52045
Fraise          Eugene     50     Ft. Madison      IA 52627
Freeman         MaryLou     5     Storm Lake       IA 50588
Gettings        Don        47     Ottumwa          IA 52501
Gronstal        Michael    42     Council Bluffs   IA 51503
Halvorson       Rod         7     Ft. Dodge        IA 50501
Hammond         Johnie     31     Ames             IA 50014
Hansen          Steven      1     Sioux City       IA 51103
Hedge           H.Kay      48     Fremont          IA 52561
Horn            Wally      27     Cedar Rapids     IA 52404
Iverson         Stewart     9     Dows             IA 50071
Jensen          John       11     Plainfield       IA 50666
```

```
Judge           Patty       46      Albia               IA 52531
Kibbie          John         4      Emmetsburg          IA 50536
King            Steve        6      Kiron               IA 51448
Kramer          Mary        37      West Des Moines     IA 50265
Lind            Jim         13      Waterloo            IA 50702
Lundby          Mary        26      Marion              IA 52302-0563
Maddox          O.Gene      38      Clive               IA 50325
McCoy           Matt        34      Des Moines          IA 50315
McKean          Andy        28      Anamosa             IA 52205
McKibben        Larry       32      Marshalltown        IA 50158
McLaren         Derryl      43      Farragut            IA 51639
Neuhauser       Mary        23      Iowa City           IA 52240
Palmer          William     33      Ankeny              IA 50021
Redfern         Don         12      Cedar Falls         IA 50613
Redwine         John         2      Sioux City          IA 51108
Rehberg         Kitty       14      Rowley              IA 52329
Rensink         Wilmer       3      Sioux Center        IA 51250
Rife            Jack        20      Durant              IA 52747
Rittmer         Sheldon     19      De Witt             IA 52742
Schuerer        Neal        30      Amana               IA 52203
Szymoniak       Elaine      36      Des Moines          IA 50310
Tinsman         Maggie      21      Davenport           IA 52807
Vilsack         Tom         49      Mt. Pleasant        IA 52641
Zieman          Lyle        16      Postville           IA 52162
```

Sample Run of `fork-5.c`*:*

```
babbage% gcc -o fork-5 fork-5.c && fork-5
The parent process continues.
The second child process is active.
The first child process is active.
Parent closing its pipe ends: parent does not use pipe
Parent waits for both children to finish.

                Members of the 1997-1998 Iowa Senate
Dearden         Dick        35      Des Moines          IA 50317
Fraise          Eugene      50      Ft. Madison         IA 52627
Freeman         MaryLou      5      Storm Lake          IA 50588
Gronstal        Michael     42      Council Bluffs      IA 51503
Halvorson       Rod          7      Ft. Dodge           IA 50501
.
.  <output listing truncated to save space>
.
Hedge           H.Kay       48      Fremont             IA 52561
Rife            Jack        20      Durant              IA 52747
Drake           Richard     24      Muscatine           IA 52761
Deluhery        Patrick     22      Davenport           IA 52804
Tinsman         Maggie      21      Davenport           IA 52807
Parent finished.
```

- The truncated listing shows that the file is listed and sorted by zip code – the contents of the 6th column.

Program 6: fork-6.c

Program reads, filters, and sorts the system user/password file.

```
/* This program reads the public user/password file, removes the username
   and owner first and last name, and orders the result by last name.

   The program is organized as follows:
      one child process uses the ypcat utility to read the user/password file
      the main program removes the needed data from each line
      a second child process sorts
   Processes are connected by pipes, which are set up using popen  */

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

#define MAX_USERNAME 10  /* maximum size allocated for username */
#define MAX_NAME 20      /* maximum size of a first or last name */

int main (void)
{  FILE *fpin, *fpout;   /* file variables for main process I/O */

   char lastname [MAX_NAME];
   char ch;              /* current character being processed */
   int i;                /* index variable */

   /* print header for output */
   printf ("Table of usernames, first names, and last names on system\n\n");
   printf ("           First     Last\n");
   printf ("Username   Name      Name\n\n");
   fflush(stdout);  /* be sure headers printed before proceeding */

   /* set up child process to read user/password file */
   fpin = popen("ypcat passwd", "r");  /* connect stdout of ypcat to fpin */
   if (fpin == NULL)
     { perror ("error in starting ypcat");
       exit(1);
     }

   /* set up child process to perform the final sort */
   fpout = popen("sort -n +2", "w");  /* connect stdin of sort to fpout */
   if (fpout == NULL)
     { perror ("error in starting sort");
       exit(1);
     }

   /* use file streams fpin, fpout as I/O for name processing */
   while ((ch = fgetc(fpin)) != EOF) /* process until end of file */
     { while (ch != '\n')    /* process line */
       { /* read username -- up to first colon : */
         for (i=0; ch != ':'; i++)  /* username is line up to 1st colon */
           { fputc(ch, fpout);
             ch = fgetc(fpin);
           }
         for ( ; i < MAX_USERNAME; i++)
            fputc(' ', fpout);  /* separate username from rest by spaces */
```

```
        /* skip 3 fields, each of which end with a colon */
        while (fgetc(fpin) != ':');
        while (fgetc(fpin) != ':');
        while (fgetc(fpin) != ':');

        /* process first name -- up to space or comma*/
        ch = fgetc(fpin);
        for (i=0; (ch != ' ') && (ch != ':')&& (ch != ','); i++)
          { fputc(ch, fpout);
            ch = fgetc(fpin);
          }
        for ( ; i < MAX_USERNAME; i++)
           fputc(' ', fpout);  /* separate username from rest by spaces */

        /* process last name -- which follows last space before colon */
        ch = fgetc(fpin);
        for (i=0; (ch != ':') && (ch != ',');)
          { if (ch == ' ')
               i = 0;       /* start last name again if space found */
            else            /* add character to last name */
               { lastname[i] = ch;
                 i++;
               }
            ch = fgetc(fpin);
          }
        for (; i<MAX_NAME-1; i++)
          lastname[i] = ' ';  /* fill name field with spaces */
        lastname[MAX_NAME-1] = '\0';   /* terminate string with null */
        fprintf (fpout, "%s\n", lastname); /* send to stdout */

        /* read rest of line */
         while ((ch = fgetc(fpin)) != '\n');
      }
    }
  /* close the current I/O stream for main, wait for each child to finish,
     and check termination status of the child */
  if (fclose (fpin))
     { perror ("error in child running ypcat");
       exit (1);
     }
  if (pclose (fpout))
     { perror ("error in child running sort");
       exit (1);
     }
  exit (0);
}
```

Annotations for `fork-6.c`*:*

- This system user/password file may be viewed by the system command `ypcat passwd`. This file contains an entry for each user in the following format:

```
walker:rBhD6XLPGEe3c:207:201:Henry M. Walker:/home/walker:/bin/csh
```

  On this line, the first entry – before the first colon – is the user name. The user's actual name appears between the fourth and fifth colons, with the first name first.

- This program program reads, filters, and sorts this file with three processes:
  - § One child process runs `ypcat passwd`, reading the file and passing the result through a pipe with the child's standard output at one end and the file descriptor `fpin` at the other end.
  - § The main program reads from the `fpin` pipe, filters reformats lines to extract username and owner name information, and writes the result to a file/pipe with file descriptor `fpout`.
  - § Another child process runs `sort`, reading from a pipe connected to file descriptor `fpout` at its input end and standard input at the child's end.
- After declaring variables, the main program prints table headers. Further, the main calls `fflush(stdout)` to guarantee that this output is printed to the screen before any further processing occurs. This guarantees the headers are printed before anything else.
- The system call `popen` performs the common details for setting up a process and pipe: creating a pipe, spawning a child with `fork`, closing the unused ends of the pipe, setting the child's input or output to standard I/O, `exec`ing a shell to execute a command, and waiting for the fork/pipe setup to finish.
  - § The first parameter to `popen` specifies the program/shell to execute.
  - § The second parameter specifies the direction of communication with the pipe. A "r" indicates the child will write to standard output, so the main (parent) can read from the pipe; A "w" indicates the child will read from standard input throught the pipe.
- The system call `pclose` performs the common finish-up details at the end of a process: the I/O stream is closed, the parent waits for the child to return, and the exit status of the child is returned.
- The line extraction within the main/parent scans subsequent characters:
  - § The username occurs first in the user/password file and is terminated by a colon.
  - § Three more fields, separated by colons, occur before the name field.
  - § Names are given with the first name and last name, in that order, although middle names sometimes occur. Names usually are terminated by a comma or colon.
  - § Miscellaneous information finishes out the line.

  Processing proceeds character-by-character and field-by-field, with the relevant data being passed on. Each name is printed in a field of a standard length, so the output will be in columns. The last name is detected by reading successive characters, although the recording of letters in this last name is reset if another space is found.

The following output is truncated to save paper.

Sample Run of `fork-6.c`*:*

```
babbage% gcc -o fork-6 fork-6.c && fork-6
Table of usernames, first names, and last names on system

           First     Last
Username   Name      Name

ackelson  Laura     Ackelson
adelbe    Arnold    Adelberg
adm       *         /var/adm
adome     Alex      Adome
aerni     John      Aerni
aggarwal  Amit      Aggarwal
akhter    K.        Akhter
.
.
.
```

Program 7: read-write-1.c

A simple readers/writers program using a one-word shared memory.

```
/* A simple readers/writers program using a one-word shared memory. */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>

#define SIZE sizeof(int)   /* size of [int] integer */
#define run_length 10       /* number of iterations in test run */

int main (void)
{  pid_t pid;              /* variable to record process id of child */
   caddr_t shared_memory;   /* shared memory base address */
   int i_parent, i_child;   /* index variables */
   int value;               /* value read by child */

   /* set up shared memory segment */
  shared_memory=mmap(0, SIZE, PROT_READ | PROT_WRITE,
                            MAP_ANONYMOUS | MAP_SHARED, -1, 0);
  if (shared_memory == (caddr_t) -1)
    { perror ("error in mmap while allocating shared memory\n");
      exit (1);
    }

   if ((pid = fork()) < 0)  /* apply fork and check for error */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)
     { /* processing for child */
       printf ("The child process begins.\n");
       for (i_child = 0; i_child < run_length; i_child++)
         { sleep(1);  /* wait for memory to be updated */
           value = *shared_memory;
           printf ("Child's report:  current value = %2d\n", value);
         }
       printf ("The child is done\n");
     }
  else
     { /* processing for parent */
       printf ("The parent process begins.\n");
       for (i_parent = 0; i_parent < run_length; i_parent++)
         { *shared_memory = i_parent * i_parent;/* square into shared memory */
           printf ("Parent's report: current index = %2d\n", i_parent );
           sleep(1);  /* wait for child to read value */
         }
       wait(pid);
       printf ("The parent is done\n");
     }
   exit (0);
}
```

Annotations for `read-write-1.c`*:*

- The `mmap` procedure (from the <sys/mman.h> library) sets up a shared memory segment and returns the base address for that segment. More generally, the `mmap` function establishes a correspondence between a specified number of bytes in the process's address space with a separate block of memory. This allocation of a new memory segment has the following form:

  ```
  base_address = mmap(0, num_bytes, protection, flags, -1, 0);
  ```

  § The second parameter, `num_bytes`, specifies the number of bytes to be allocated for the new segment.
  § The third parameter, `protection`, specifies whether the segment may be used for reading, writing, executing, or other purpose. The various options are:

  `PROT_READ` – page can be read
  `PROT_WRITE` – page can be written
  `PROT_EXEC` – page can be executed
  `PROT_NONE` – page cannot be accessed

  For typical shared memory, both read and write permission is specified using the combination `PROT_READ | PROT_WRITE` .
  § The fourth parameter indicates further details of file allocation and file sharing. In `read-write-1.c`, the combination `MAP_ANONYMOUS | MAP_SHARED` indicates a new memory segment should be allocation (rather than allocating space from a file descriptor) and all writes to the memory segment should be shared with other processes.
  § The values of the other parameters provide flexibility for many applications. While the general use of `mmap` is beyond the scope of this introduction, simple allocation of a main memory segment requires a 0 value in the first and last paramter. The value -1 in the next-to-last parameter indicates a new, internal file descriptor is needed – the segment will not be part of an existing file.
  § Procedure `mmap` returns the base address (starting location) of the memory segment.

- Once memory is allocated with `mmap`, and the base address `shared_memory` is known, then the `shared_memory` variable may be used as any address within a C program. In particular, `*shared_memory` specifies the value stored at that address.

- In this program, the parent process writes successive data values to the shared memory, and the child process reads successive data values. The `sleep` statements are added in an attempt to synchronize this data transfer. Once the parent puts a value (a square of an integer) into shared memory, it sleeps for a second to allow the child time to access that value. Similarly, a child delays its processing by one second to give the parent time to put a new value into shared memory.

Sample Run of `read-write-1.c`*:*

```
babbage% gcc -o read-write-1 read-write-1.c && read-write-1
The parent process begins.
Parent's report: current index =  0
The child process begins.
Parent's report: current index =  1
Child's report:  current value =  1
Parent's report: current index =  2
Child's report:  current value =  4
Parent's report: current index =  3
Child's report:  current value =  9
Parent's report: current index =  4
Child's report:  current value = 16
Parent's report: current index =  5
Child's report:  current value = 25
```

```
Parent's report: current index =  6
Child's report:  current value = 36
Parent's report: current index =  7
Child's report:  current value = 49
Parent's report: current index =  8
Child's report:  current value = 64
Parent's report: current index =  9
Child's report:  current value = 81
Child's report:  current value = 81
The child is done
The parent is done
```

- As hoped, the `sleep` statements provide adequate time for the parent to write and the child to read, before new values are put into the shared memory.
- However, if the `sleep` statement is removed from the parent (but retained in the child), then the all values are written by the parent before any are retrieved by the child. This is shown in the following sample run, where the child gets only the last value written by the parent – the other values are lost.

Sample Run of `read-write-1.c`*:*

```
babbage% gcc -o read-write-1 read-write-1.c && read-write-1
The parent process begins.
Parent's report: current index =  0
Parent's report: current index =  1
Parent's report: current index =  2
Parent's report: current index =  3
Parent's report: current index =  4
Parent's report: current index =  5
Parent's report: current index =  6
Parent's report: current index =  7
Parent's report: current index =  8
Parent's report: current index =  9
The child process begins.
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
Child's report:  current value = 81
The child is done
The parent is done
```

Sample Run of `read-write-1.c`*:*

```
Script started on Mon Sep 28 14:51:27 1998
babbage% gcc -o read-write-1 read-write-1.c && read-write-1
The child process begins.
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
Child's report:  current value =  0
The child is done
The parent process begins.
Parent's report: current index =  0
Parent's report: current index =  1
Parent's report: current index =  2
Parent's report: current index =  3
Parent's report: current index =  4
Parent's report: current index =  5
Parent's report: current index =  6
Parent's report: current index =  7
Parent's report: current index =  8
Parent's report: current index =  9
The parent is done
babbage%
script done on Mon Sep 28 14:52:01 1998
```

- If the `sleep` statement is removed from the child (but retained in the parent), then the child performs all of its reading before the parent makes any updates. Thus, most data from the parent arrive too late to be used by the child.

Program 8: read-write-2.c

A readers/writers program, using an intermediate buffer of 5 integer words, plus 2 index markers.

```
/* A simple readers/writers program using a shared buffer and spinlocks  */

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

#define BUF_SIZE 5            /* logical size of buffer */
#define SHARED_MEM_SIZE (BUF_SIZE+2)*sizeof(int) /* size of shared memory */
#define run_length 10  /* number of iterations in test run */

int main (void)
{  pid_t pid;          /* variable to record process id of child */
   caddr_t shared_memory;   /* shared memory base address */

   int *in;         /* pointer to logical 'in' address for writer */
   int *out;        /* pointer to logical 'out' address for reader */
   int *buffer;       /* logical base address for buffer */

   int i_child, j_child;   /* index variables */
   int value;              /* value read by child */

   /* set up shared memory segment */
   shared_memory=mmap(0, SHARED_MEM_SIZE, PROT_READ | PROT_WRITE,
                            MAP_ANONYMOUS | MAP_SHARED, -1, 0);
   if (shared_memory == (caddr_t) -1)
     { perror ("error in mmap while allocating shared memory\n");
       exit (1);
     }

   /* shared memory segment will be organized as follows:
       0                                                   n-1 n  n+1
      ---------------------------------------------------------------
      |                                                       |  |  |
      ---------------------------------------------------------------
       ^                                                       ^  ^
      buffer                                                   in out
   */

   buffer = (int*) shared_memory; /* logical buffer starts at shared segment */
   in  = (int*) shared_memory + BUF_SIZE*sizeof(int);
   out = (int*) shared_memory + (BUF_SIZE+1)*sizeof(int);

   *in = *out = 0;          /* initial starting points */

   if (-1 == (pid = fork())) /* check for error in spawning child process */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)
     { /* processing for child == reader */
```

```
      printf ("The reader process begins.\n");

      for (i_child = 0; i_child < run_length; i_child++)
         {  while (*in == *out) ;       /* spinlock waiting for data */
            value = buffer[*out];
            *out = (*out + 1) % BUF_SIZE;
            printf ("Reader's report: item %2d == %2d\n", i_child, value);
        }
      printf ("Reader done.\n");
    }
  else
    { /* processing for parent == writer */
      printf ("The writer process begins.\n");

      for (j_child = 0; j_child < run_length; j_child++)
        {  while ((*in + 1) % BUF_SIZE == *out);/* spinlock waiting for space */
           buffer[*in] = j_child*j_child;    /* put data in buffer */
           *in = (*in + 1) % BUF_SIZE;
           printf ("Writer's report: item %2d put in buffer\n", j_child);
         }
      wait (pid);
      printf ("Writer done.\n");
    }
   exit (0);
}
```

Annotations for `read-write-2.c`*:*

- This program follows the outline for a bounded buffer solution to the readers/writers problem, as given in Silberschatz and Galvin, *OperatingSystems Concepts*, *Fifth Edition*, page 102.
- A logical `buffer` is allocated in shared memory, and buffer indexes, `in` and `out`, are used identify where data will be stored or read by the writer or reader process. More specifically,
  § `*in` gives the next free place in the `buffer` for the writer to enter data.
  § `*out` gives the first place in the `buffer` for the reader to extract data.
- Writing to the `buffer` may continue unless the buffer is full (i.e., `(*in + 1) % BUF_SIZE == *out`) and reading from the buffer may proceed unless the buffer is empty (i.e., `*in == *out`). Both conditions are tested in spin locks.
- `shared_memory` must be large enough for the `buffer` and the variables `in` and `out`. As integers are being stored in the `in` and `out`, this implies shared memory must be 2 integers larger than the buffer of *BUF_SIZE* integers.
  § In the program, the `buffer` occupies the first n words of `shared_memory`.
  § `in` gives the address of word `BUF_SIZE+1` of `shared_memory`.
  § `out` gives the address of word `BUF_SIZE+2` of `shared_memory`.
- Since `buffer` represents the base address of an array of integers, it is declared as a pointer to an integer. Similarly, `in` and `out` specify addresses of integers, so `*in` and `*out` specify the values stored at those addresses.
- The `buffer` array starts at the beginning of the `shared_memory` segment. Thus, `buffer` may be initialized with the address of `shared_memory`, where the cast `(int*)` translates from the address type `caddr_t` to the more familiar integer-addressing format.

- `in` and `out` are initialized by computing the appropriate address within `shared_memory`. In each case, address type `caddr_t` is converted to an integer form. For `in`, the address will start after the `buffer` array – `BUF_SIZE` words from the start of `shared_memory`. The address for `out` begins one word later.
- The starting buffer indices, `*in` and `*out`, identify the beginning of the buffer – at position 0.
- The writer places a value in `buffer[*in]` and increments the buffer array index. (For simplicity, the value produced is just the square of a sequence number.)
- The reader retrieves the value from `buffer[*out]` and also increments the buffer array index.
- Following good practice, the parent waits for the child process to finish at the end.

Sample Run of `read-write-2.c`*:*

```
babbage% gcc -o read-write-2 read-write-2.c && read-write-2
The reader process begins.
The writer process begins.
Writer's report: item  0 put in buffer
Writer's report: item  1 put in buffer
Writer's report: item  2 put in buffer
Writer's report: item  3 put in buffer
Reader's report: item  0 ==  0
Reader's report: item  1 ==  1
Reader's report: item  2 ==  4
Reader's report: item  3 ==  9
Writer's report: item  4 put in buffer
Writer's report: item  5 put in buffer
Writer's report: item  6 put in buffer
Writer's report: item  7 put in buffer
Reader's report: item  4 == 16
Reader's report: item  5 == 25
Reader's report: item  6 == 36
Reader's report: item  7 == 49
Writer's report: item  8 put in buffer
Writer's report: item  9 put in buffer
Reader's report: item  8 == 64
Reader's report: item  9 == 81
Reader done.
Writer done.
```

- While the reader process starts, it then waits until the writer puts data in the buffer. However, the writer is blocked once the buffer is full (with 4 items – the writer's test guarantees that one slot in the buffer will remain empty).

Program 9: read-write-3.c

A readers/writers program with a shared buffer and semaphores to coordinate buffer access.

```
/* A readers/writers program using a shared buffer and semaphores  */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/sem.h>

#define BUF_SIZE 5            /* logical size of buffer */
#define SHARED_MEM_SIZE (BUF_SIZE+2)*sizeof(int) /* size of shared memory */
#define run_length 10  /* number of iterations in test run */

#define buf_used 0     /* semaphore array index to check buffer elts used */
#define buf_space 1    /* semaphore array index to check buffer elts empty */

int sem_init(void)
{  /* procedure to create and initialize semaphores and return semaphore id,
      assuming two semaphores defined in the given array of semaphores     */
   int semid;

   /* create new semaphore set of 2 semaphores */
   if ((semid = semget (IPC_PRIVATE, 2, IPC_CREAT | 0600)) < 0)
     {  perror ("error in creating semaphore");/* 0600 = read/alter by user */
        exit (1);
     }

   /* initialization of semaphores */
   /* BUF_SIZE free spaces in empty buffer */
   if (semctl (semid, buf_space, SETVAL, BUF_SIZE) < 0)
     {  perror ("error in initializing first semaphore");
        exit (1);
     }

   /* 0 items in empty buffer */
   if (semctl (semid, buf_used, SETVAL, 0) < 0)
     {  perror ("error in initializing second semaphore");
        exit (1);
     }
   return semid;
}

void P(int semid, int index)
{/* procedure to perform a P or wait operation on a semaphore of given index */
  struct sembuf sops[1];  /* only one semaphore operation to be executed */

   sops[0].sem_num = index;/* define operation on semaphore with given index */
   sops[0].sem_op  = -1;   /* subtract 1 to value for P operation */
   sops[0].sem_flg = 0;    /* type "man semop" in shell window for details */

   if (semop (semid, sops, 1) == -1)
     {  perror ("error in semaphore operation");
        exit (1);
     }
}
```

```
void V(int semid, int index)
{/* procedure to perform a V or signal operation on semaphore of given index */
   struct sembuf sops[1];  /* define operation on semaphore with given index */

   sops[0].sem_num = index;/* define operation on semaphore with given index */
   sops[0].sem_op  = 1;    /* add 1 to value for V operation */
   sops[0].sem_flg = 0;    /* type "man semop" in shell window for details */

   if (semop (semid, sops, 1) == -1)
     {  perror ("error in semaphore operation");
        exit (1);
     }
}

int main (void)
{  pid_t pid;          /* variable to record process id of child */

   /* shared memory elements */
   caddr_t shared_memory;   /* shared memory base address */
   int *in;         /* pointer to logical 'in' address for writer */
   int *out;        /* pointer to logical 'out' address for reader */
   int *buffer;     /* logical base address for buffer */

   /* semaphore elements */
   int semid;       /* identifier for a semaphore set */

   /* local variables */
   int i_child, j_child;   /* index variables */
   int value;              /* value read by child */

   /* set up shared memory segment */
   shared_memory=mmap(0, SHARED_MEM_SIZE, PROT_READ | PROT_WRITE,
                           MAP_ANONYMOUS | MAP_SHARED, -1, 0);
   if (shared_memory == (caddr_t) -1)
     { perror ("error in mmap while allocating shared memory\n");
       exit (1);
     }

   /* set up pointers to appropriate places in shared memory segment */
   buffer = (int*) shared_memory; /* logical buffer starts at shared segment */
   in  = (int*) shared_memory + BUF_SIZE*sizeof(int);
   out = (int*) shared_memory + (BUF_SIZE+1)*sizeof(int);

   *in = *out = 0;          /* initial starting points */

   /* create and initialize semaphore */
   semid = sem_init();

   if (-1 == (pid = fork())) /* check for error in spawning child process */
     { perror ("error in fork");
       exit (1);
     }

   if (0 == pid)
     { /* processing for child == reader */
       printf ("The reader process begins.\n");
```

```
      for (i_child = 0; i_child < run_length; i_child++)
        {  P(semid, buf_used);  /* wait semaphore for something used */
           value = buffer[*out];
           *out = (*out + 1) % BUF_SIZE;
           printf ("Reader's report: item %2d == %2d\n", i_child, value);
           V(semid, buf_space); /* signal semaphore for space available */
           if ((i_child % 3) == 1)
             sleep(1);  /* take time to process every third element */
        }
      printf ("Reader done.\n");
    }
  else
    { /* processing for parent == writer */
      printf ("The writer process begins.\n");

      for (j_child = 0; j_child < run_length; j_child++)
        {  P(semid, buf_space);/* wait semaphore for space available */
           buffer[*in] = j_child*j_child;    /* put data in buffer */
           *in = (*in + 1) % BUF_SIZE;
           printf ("Writer's report: item %2d put in buffer\n", j_child);
           V(semid, buf_used); /* signal semaphore for something used */
           if ((j_child % 4) == 0)
             sleep(1); /* take time to generate every fourth element */
         }
      wait (pid);
      printf ("Writer done.\n");

      /* Remove the semaphore from the system and destroy the set of
         semaphores and data structure associated with it. */
      if (semctl (semid, 0, IPC_RMID) < 0)
        {  perror ("error in removing semaphore from the system");
           exit (1);
        }
      printf ("Semaphore cleanup complete.\n");
    }
   exit (0);
}
```

Annotations for `read-write-3.c`*:*

- This program follows the same approach to a buffer in shared memory as the previous example. `shared_memory` is declared and allocated with `mmap`, and variables `buffer`, `in` and `out` reference various parts of that shared memory.
- Writing to and reading from the buffer are protected by semaphores. Conceptually, the `buf_used` keeps track of how many buffer elements are used, while `buf_space` keeps track of how many empty spaces remain in the buffer.
- Practically, semaphores in Unix [GNU] C are part of the <sys/sem.h> library. All semaphores are considered as part of a set or array and accessed through an integer identification number (`semid` in this program). Work with semaphores follows three main steps:
  § A set or array of semaphores are created with `semget` .
  § Semaphores are initialized or otherwise controled by `setctl` .
  § Semaphore operations are performed using `semop` .
- To clarify the logic of the program, these tasks are placed in separate, helping procedures.

- In the main program, the statement `semid = sem_init()` creates and initializes a semaphore, and the semaphore's id is stored in `semid` .
  - § As all semaphores in Unix [GNU] C are declared in arrays, this program defines index `buf_used` for semaphore 0 – the semaphore which will be used to keep track how much space within the buffer has been used.
  - § Similarly, this program defines index `buf_space` for semaphore 1 – the semaphore which will be used to keep track how much empty space exists within the buffer.
- Following standard operating system terminology, the operation `P(semid, buf_used)` causes the process to wait if no space is used. Similarly, `P(semid, buf_space)` causes the process to wait if no empty space remains. If the counts for these variables are nonzero, then the respective processes continue, decrementing the number of used items or the amount of space by 1. The alternative name for the `P` operation is `wait`.
- Also using standard terminology, the operation `V(semid, buf_used)` increments the semaphore's count of used elements by increased by 1. Similarly, `V(semid, buf_space)` increases the count of free space by 1. A common alternative name to the `V` operation is `signal` .
- The parent/writer process continues to put data within the buffer as long as space remains. The statement `P(semid, buf_space)` prevents the writer from proceeding if the buffer is full (i.e., no space remains). After the writer places data in the buffer, the statement `V(semid, buf_space)` updates the count of free space.
- The child/reader process uses the statement `P(semid, buf_used)` to wait until the buffer is used to store some relevant data. After reading the data, the statement `V(semid, buf_space)` indicates that one space within the buffer has become free.
- In the program, the `sleep` statements in both the parent and child are added only to demonstrate different times of processing, making the output more interesting.
- Semaphore creation within procedure `sem_init` utilizes the statement
  `semget (IPC_PRIVATE, 2, IPC_CREAT | 0600)`
  - § `IPC_PRIVATE` indicates a private semaphore set is to be referenced, and the parameter 2 indicates this set will contain two semaphores.
  - § The last parameter `IPC_CREAT | 0600` indicates this semaphore set is to be created and the user processes should have read/write access (code 0600).
  - § `semget` normally returns a new semaphore identification integer, although a negative response indicates an error has occurred.
- Semaphore initialization involves setting specific values for each semaphore with the `semctl` statement.
  - § The first parameter identifies the semaphore set, while the second parameter indicates which element of that set is being referenced.
  - § The third parameter indicates the operation to be performed. `SETVAL` sets the specific semaphore to the value that follows (e.g., `BUF_SIZE` or 0), while `IPC_RMID` removes the entire semaphore from the system. Another command option, `SETALL`, sets all semaphores to the value that follows.
- The procedure `semop` is used to change the value of a semaphore.
  - § The first parameter one specifies which semaphore set is to be modified.
  - § The second parameter gives an array of commands. As shown within the program's `P` and `V` procedures, a command involves setting three fields within a `sembuf` structure (e.g, `sops` in this program). `sem_num` indicates which semaphore in the set to change. `sem_op` indicates the operation, such as add or subtract 1 from the semaphore's value. Addition always proceeds as expected, while subtraction may result in blocking the process if the current value already is 0. For most purposes, setting `sem_flg` to 0 provides an appropriate flag value.
  - § The final parameter to `semop` gives the number of operations to be applied to the semaphore. In both the `P` and `V` operations, only a single command is given. Thus, the `sembuf` array `sops` has size 1, and the last parameter to `semop` is 1.

Sample Run of `read-write-3.c`*:*

```
babbage% gcc -o read-write-3 read-write-3.c && read-write-3
The reader process begins.
The writer process begins.
Writer’s report: item  0 put in buffer
Reader’s report: item  0 ==  0
Writer’s report: item  1 put in buffer
Reader’s report: item  1 ==  1
Writer’s report: item  2 put in buffer
Writer’s report: item  3 put in buffer
Writer’s report: item  4 put in buffer
Reader’s report: item  2 ==  4
Writer’s report: item  5 put in buffer
Writer’s report: item  6 put in buffer
Writer’s report: item  7 put in buffer
Reader’s report: item  3 ==  9
Writer’s report: item  8 put in buffer
Reader’s report: item  4 == 16
Writer’s report: item  9 put in buffer
Reader’s report: item  5 == 25
Reader’s report: item  6 == 36
Reader’s report: item  7 == 49
Reader’s report: item  8 == 64
Reader’s report: item  9 == 81
Reader done.
Writer done.
Semaphore cleanup complete.
babbage% read-write-3
The reader process begins.
The writer process begins.
Writer’s report: item  0 put in buffer
Reader’s report: item  0 ==  0
Writer’s report: item  1 put in buffer
Reader’s report: item  1 ==  1
Writer’s report: item  2 put in buffer
Writer’s report: item  3 put in buffer
Writer’s report: item  4 put in buffer
Reader’s report: item  2 ==  4
Reader’s report: item  3 ==  9
Reader’s report: item  4 == 16
Writer’s report: item  5 put in buffer
Writer’s report: item  6 put in buffer
Writer’s report: item  7 put in buffer
Writer’s report: item  8 put in buffer
Reader’s report: item  5 == 25
Reader’s report: item  6 == 36
Reader’s report: item  7 == 49
Writer’s report: item  9 put in buffer
Reader’s report: item  8 == 64
Reader’s report: item  9 == 81
Reader done.
Writer done.
Semaphore cleanup complete.
```

- The two sample runs show several results of the reader and writer processes working independently. Sometimes, one process must wait for the other, while they both can work concurrently at other times.
- At the conclusion of the program, the parent waits for its child, and then the semaphore set is removed from the system. While normal program termination should clean up the system's semaphore table, this deallocation step provides additional assurance that semaphores will not accumulate after a program terminates.

Program 10: read-write-4.c

A readers/writers program with multiple readers and multiple writers communicating through a shared buffer, with synchronization achieved through semaphores.

```
/* A readers/writers program for multiple readers and multiple writers  */
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
#include <sys/mman.h>
#include <sys/sem.h>

#define NUM_READERS 5  /* number of reader processes to be spawned */
#define NUM_WRITERS 6  /* number of writer processes to be spawned */

#define BUF_SIZE 5            /* logical size of buffer */
#define SHARED_MEM_SIZE (BUF_SIZE+2)*sizeof(int) /* size of shared memory */
#define reader_length 12  /* number of iterations for reader in test run */
#define writer_length 10  /* number of iterations for writer in test run */

#define buf_used 0     /* semaphore array index to check buffer elts used */
#define buf_space 1    /* semaphore array index to check buffer elts empty */
#define mutex 2        /* semaphore index for mutual exclusion to buffer*/

int sem_create(int num_semaphores)
{  /* procedure to create specified number of semaphores */
   int semid;

   /* create new semaphore set of semaphores */
   if ((semid = semget (IPC_PRIVATE, num_semaphores, IPC_CREAT | 0600)) < 0)
     {  perror ("error in creating semaphore");/* 0600 = read/alter by user */
        exit (1);
     }
   return semid;
}

void sem_init(int semid, int index, int value)
{  /* procedure to initialize specified semaphore to given value */
   if (semctl (semid, index, SETVAL, value) < 0)
     {  perror ("error in initializing first semaphore");
        exit (1);
     }
}

void P(int semid, int index)
{/* procedure to perform a P or wait operation on a semaphore of given index */
  struct sembuf sops[1];  /* only one semaphore operation to be executed */

   sops[0].sem_num = index;/* define operation on semaphore with given index */
   sops[0].sem_op  = -1;   /* subtract 1 to value for P operation */
   sops[0].sem_flg = 0;    /* type "man semop" in shell window for details */

   if (semop (semid, sops, 1) == -1)
     {  perror ("error in semaphore operation");
        exit (1);
     }
}
```

```
void V(int semid, int index)
{/* procedure to perform a V or signal operation on semaphore of given index */
   struct sembuf sops[1];  /* define operation on semaphore with given index */

   sops[0].sem_num = index;/* define operation on semaphore with given index */
   sops[0].sem_op  = 1;    /* add 1 to value for V operation */
   sops[0].sem_flg = 0;    /* type "man semop" in shell window for details */

   if (semop (semid, sops, 1) == -1)
     {  perror ("error in semaphore operation");
        exit (1);
     }
}

int main (void)
{  pid_t pid;          /* variable to record process id of child */

   /* shared memory elements */
   caddr_t shared_memory;   /* shared memory base address */
   int *in;         /* pointer to logical 'in' address for writer */
   int *out;        /* pointer to logical 'out' address for reader */
   int *buffer;     /* logical base address for buffer */

   /* semaphore elements */
   int semid;       /* identifier for a semaphore set */

   /* record of spawned processes */
   pid_t proc[NUM_READERS+NUM_WRITERS];

   /* local variables */
   int p_count, i;  /* index variables */
   int value;       /* value read by child */

   /* set up shared memory segment */
   shared_memory=mmap(0, SHARED_MEM_SIZE, PROT_READ | PROT_WRITE,
                           MAP_ANONYMOUS | MAP_SHARED, -1, 0);
   if (shared_memory == (caddr_t) -1)
     { perror ("error in mmap while allocating shared memory\n");
       exit (1);
     }

   /* set up pointers to appropriate places in shared memory segment */
   buffer = (int*) shared_memory; /* logical buffer starts at shared segment */
   in  = (int*) shared_memory + BUF_SIZE*sizeof(int);
   out = (int*) shared_memory + (BUF_SIZE+1)*sizeof(int);

   *in = *out = 0;          /* initial starting points */

   /* create and initialize semaphores */
   semid = sem_create(3);
   sem_init(semid, buf_used, 0);
   sem_init(semid, buf_space, BUF_SIZE);
   sem_init(semid, mutex, 1);

   /* spawn writer processes */
   for (p_count = 1; p_count <= NUM_WRITERS; p_count++)
```

```
  { if (-1 == (pid = fork())) /* spawn child process */
     { perror ("error in fork");
       exit (1);
     }

    if (0 == pid)
     { /* processing for parent == writer */
       printf ("The writer process %d begins.\n", p_count);

       for (i = 0; i < writer_length; i++)
         {value = 100*p_count + i;/* writer == first digit of value */
          P(semid, buf_space);/* wait semaphore for space available */
          P(semid, mutex);    /* wait semaphore for buffer access */
          buffer[*in] = value;/* put data in buffer */
          *in = (*in + 1) % BUF_SIZE;
          V(semid, mutex);    /* signal semaphore for buffer access */
          V(semid, buf_used); /* signal semaphore for something used */
          /*printf ("Writer %d: item %2d put %d in buffer\n", j_child);*/
         }
       printf ("Writer %d done.\n", p_count);
       exit(0);
     }
    else proc[p_count-1] = pid;
  }

/* spawn reader processes */
for (p_count = 1; p_count <= NUM_READERS; p_count++)
  { if (-1 == (pid = fork())) /* spawn child process */
     { perror ("error in fork");
       exit (1);
     }

    if (0 == pid)
      { /* processing for child == reader */
        printf ("The reader process %d begins.\n", p_count);

        for (i = 0; i < reader_length; i++)
          {P(semid, buf_used);  /* wait semaphore for something used */
           P(semid, mutex);     /* wait semaphore for buffer access */
           value = buffer[*out];/* take data from buffer */
           *out = (*out + 1) % BUF_SIZE;
           V(semid, mutex);     /* signal semaphore for buffer access */
           V(semid, buf_space); /* signal semaphore for space available */
           printf ("Reader %d: item %2d == %2d\n", p_count, i, value);
           if ((i+p_count)%5 == 0) /* pause somewhere in processing */
             sleep(1);             /* to make output more interesting */
          }
        printf ("Reader %d done.\n", p_count);
        exit(0);
      }
    else proc[p_count+NUM_READERS-1] = pid;
  }

/* wait for all children to finish */
printf("All child processes spawned by parent\n");
printf("Parent waiting for children to finish\n");
```

```
    for (p_count = 0; p_count<NUM_WRITERS+NUM_READERS; p_count++)
      waitpid(proc[p_count], NULL, 0);

    /* Remove the semaphore from the system and destroy the set of
       semaphores and data structure associated with it. */
    if (semctl (semid, 0, IPC_RMID) < 0)
      {  perror ("error in removing semaphore from the system");
         exit (1);
      }
    printf ("Semaphore cleanup complete.\n");

    exit (0);
}
```

Annotations for `read-write-4.c`*:*

- This program utilizes a shared buffer following the same shared-memory facilities (with `mmap`) used in the previous programs.
- This program uses three semaphores, one for the amount of buffer space used, one for the amount of buffer space free, and one to guarantee that only one process actually accesses the buffer at a time. As in the previous program, these semaphores are declared in one semaphore set, and constants (`buf_used`, `buf_space`, and `mutex`) are defined to help remember which semaphore index is which.
- Semaphores are created in procedure `sem_create`, which calls the system procedure `semget`, performs appropriate error checking, and returns the id of the semaphore set.
- A semaphore is initialized in procedure `sem_init`, which sets a specific semaphore to a given value. While `sem_init` simply calls `semctl` with an extra `SETVAL` parameter, the separation of the error checking from other processing makes the main program cleaner.
- The semaphore `P` and `V` operations used here follow the code from the previous program.
- The main program spawns `NUM_WRITERS` processes for writing and saves the id's in an array `proc`. Then the main program spawns `NUM_READERS` processes for reading and adds the id's to the process id array `proc`.
- As in the previous program, a writer process must wait to write until there is space in the buffer, and this is checked by the `buf_space` semaphore. Similarly, a reader process cannot take data out of an empty buffer, and this is checked by the `buf_used` semaphore.
- Since multiple writers and multiple readers may want to access the buffer at the same time, there is a possibility that two unconstrained processes might attempt to read or write from the same place in the buffer. This concurrent buffer access is prevented by the `mutex` semaphore, which guarantees that only one process can actually use the buffer at a time.
- While the writers do not display their data directly on the output, the data are coded, so the first digit of a value indicates the writer process number.
- As in the previous program, a `sleep` statement is added to the reader processes, so that various processes are blocked at various times. This seems to make the scheduling of processes more interesting.
- After the main program spawns all the writer and reader processes, it waits until all spawned processes terminate. This is done by keeping the child process ids in an array and then performing a `wait` for each of these ids in turn.

Sample Run of `read-write-4.c`*:*

```
babbage% gcc -o read-write-4 read-write-4.c && read-write-4
The writer process 1 begins.
The writer process 2 begins.
The writer process 4 begins.
The writer process 3 begins.
The writer process 5 begins.
The writer process 6 begins.
The reader process 2 begins.
Reader 2: item  0 == 100
Reader 2: item  1 == 101
Reader 2: item  2 == 102
Reader 2: item  3 == 103
The reader process 1 begins.
Reader 1: item  0 == 104
Reader 1: item  1 == 105
Reader 1: item  2 == 200
Reader 1: item  3 == 400
Reader 1: item  4 == 300
The reader process 3 begins.
Reader 3: item  0 == 500
Reader 3: item  1 == 600
Reader 3: item  2 == 106
The reader process 4 begins.
Reader 4: item  0 == 201
Reader 4: item  1 == 401
The reader process 5 begins.
Reader 5: item  0 == 301
All child processes spawned by parent
Parent waiting for children to finish
Reader 2: item  4 == 501
Reader 1: item  5 == 601
Reader 1: item  6 == 107
Reader 1: item  7 == 202
Reader 1: item  8 == 402
Reader 1: item  9 == 302
Reader 4: item  2 == 602
Reader 4: item  3 == 108
Writer 1 done.
Reader 4: item  4 == 203
Reader 4: item  5 == 403
Reader 3: item  3 == 502
Reader 3: item  4 == 503
Reader 3: item  5 == 603
Reader 3: item  6 == 109
Reader 3: item  7 == 204
Reader 4: item  6 == 303
Reader 5: item  1 == 404
Reader 5: item  2 == 405
Reader 5: item  3 == 504
Reader 5: item  4 == 604
Reader 5: item  5 == 205
Reader 2: item  5 == 406
Reader 2: item  6 == 304
Reader 2: item  7 == 505
Reader 2: item  8 == 605
Reader 1: item 10 == 206
Reader 4: item  7 == 305
Reader 4: item  8 == 506
Reader 4: item  9 == 606
Reader 4: item 10 == 207
Writer 4 done.
Reader 4: item 11 == 408
Reader 3: item  8 == 407
Reader 3: item  9 == 306
Reader 3: item 10 == 507
Reader 3: item 11 == 607
Reader 3 done.
Writer 2 done.
Reader 5: item  6 == 208
Reader 5: item  7 == 409
Writer 5 done.
Reader 5: item  8 == 307
Writer 6 done.
Reader 5: item  9 == 508
Writer 3 done.
Reader 5: item 10 == 608
Reader 2: item  9 == 209
Reader 2: item 10 == 308
Reader 2: item 11 == 509
Reader 2 done.
Reader 1: item 11 == 609
Reader 1 done.
Reader 4 done.
Reader 5: item 11 == 309
Reader 5 done.
Semaphore cleanup complete.
```

- In this output, the interleaving of the writers may be inferred, in that readers take data from the buffer in the order the data are written, and the writer of each data item is known by the hundred's digit of that data.