0.0.1 Semaphores for Threads

- If the threads work too quickly, the queue of jobs will empty and the threads will exit.
- What we might like instead is a mechanism for blocking the threads when the queue empties until new jobs become available.
- A semaphore is a counter (non-negative integer) that can be used to synchronize multiple threads. As with a mutex, Linux guarantees that checking or modifying the value of a semaphore can be done safely, without creating a race condition.
- A semaphore supports two basic operations:
  - A **wait** (down) operation decrements the value of the semaphore by 1. If the value is already zero, the operation blocks until the value of the semaphore becomes positive (due to the action of some other thread). When the semaphore's value becomes positive, it is decremented by 1 and the wait operation returns.
  - A **post** (up) operation increments the value of the semaphore by 1. If the semaphore was previously zero and other threads are blocked in a wait operation on that semaphore, one of those threads is unblocked and its wait operation completes (which brings the semaphore's value back to zero).
- A semaphore is represented by a **sem_t** variable.
  - initialize it using the **sem_init** function, passing a pointer to the **sem_t** variable.
  - the second parameter should be zero.
  - the third parameter is the semaphore's initial value.
  - to deallocate it with **sem_destroy**.
  - to wait on a semaphore, use **sem_wait**.
  - to post to a semaphore, use **sem_post**.
  - a nonblocking wait function, **sem_trywait**, is also provided.
  - to retrieve the current value of a semaphore, **sem_getvalue**, (to do this could lead to a race condition).
- The following code (see Fig. 0.0.1) controls the queue with a semaphore.

```
#include <malloc.h>
#include <pthread.h>
#include <semaphore.h>
struct job {
  /* Link field for linked list.  */
  struct job* next;
  /* Other fields describing work to be done... */
};
/* A linked list of pending jobs.  */
struct job* job_queue;
extern void process_job (struct job*);
/* A mutex protecting job_queue.  */
pthread_mutex_t job_queue_mutex = PTHREAD_MUTEX_INITIALIZER;
/* A semaphore counting the number of jobs in the queue.  */
sem_t job_queue_count;
/* Perform one-time initialization of the job queue.  */
void initialize_job_queue ()
{
  /* The queue is initially empty.  */
  job_queue = NULL;
  /* Initialize the semaphore which counts jobs in the queue.  Its
     initial value should be zero.  */
  sem_init (&job_queue_count, 0, 0);
}
/* Process queued jobs until the queue is empty.  */
void* thread_function (void* arg)
{
  while (1) {
    struct job* next_job;
    /* Wait on the job queue semaphore.  If its value is positive,
       indicating that the queue is not empty, decrement the count by
       one.  If the queue is empty, block until a new job is enqueued.  */
    sem_wait (&job_queue_count);
    /* Lock the mutex on the job queue.  */
    pthread_mutex_lock (&job_queue_mutex);
    /* Because of the semaphore, we know the queue is not empty.  Get
       the next available job.  */
    next_job = job_queue;
    /* Remove this job from the list.  */
    job_queue = job_queue->next;
    /* Unlock the mutex on the job queue, since we're done with the
       queue for now.  */
    pthread_mutex_unlock (&job_queue_mutex);
    /* Carry out the work.  */
    process_job (next_job);
    /* Clean up.  */
    free (next_job);
  }
  return NULL;
}
/* Add a new job to the front of the job queue.  */
void enqueue_job (/* Pass job-specific data here...  */)
{
  struct job* new_job;
  /* Allocate a new job object.  */
  new_job = (struct job*) malloc (sizeof (struct job));
  /* Set the other fields of the job struct here...  */
  /* Lock the mutex on the job queue before accessing it.  */
  pthread_mutex_lock (&job_queue_mutex);
  /* Place the new job at the head of the queue.  */
  new_job->next = job_queue;
  job_queue = new_job;
  /* Post to the semaphore to indicate another job is available.  If
     threads are blocked, waiting on the semaphore, one will become
     unblocked so it can process the job.  */
  sem_post (&job_queue_count);
  /* Unlock the job queue mutex.  */
  pthread_mutex_unlock (&job_queue_mutex);
}
```

Figure 1: Job Queue Controlled by a Semaphore

- If the semaphore's value is zero, indicating that the queue is empty, the thread will simply block until the semaphore s value becomes positive, indicating that a job has been added to the queue.

0.1 Linux Thread Implementation

- Whenever you call **pthread_create** to create a new thread, Linux creates a new process that runs that thread.
- However, this process is not the same as a process you would create with fork; in particular, it shares the same address space and resources as the original process rather than receiving copies.
- The manager thread is created the first time a program calls **pthread_create** to create a new thread. thread-pid.c (see Fig. 0.1)

```
#include <pthread.h>
#include <stdio.h>
#include <unistd.h>
void* thread_function (void* arg)
{
  fprintf (stderr, "child thread pid is %d\n", (int) getpid ());
  /* Spin forever.  */
  while (1);
  return NULL;
}
int main ()
{
  pthread_t thread;
  fprintf (stderr, "main thread pid is %d\n", (int) getpid ());
  pthread_create (&thread, NULL, &thread_function, NULL);
  /* Spin forever.  */
  while (1);
  return 0;
}
```

Figure 2: Print Process IDs for Threads

0.2 Processes Vs. Threads

For some programs that benefit from concurrency, the decision whether to use processes or threads can be difficult.

- All threads in a program must run the same executable. A child process, on the other hand, may run a different executable by calling an **exec** function.

- An errant thread can harm other threads in the same process because threads share the same virtual memory space and other resources. For instance, a wild memory write through an uninitialized pointer in one thread can corrupt memory visible to another thread. An errant process, on the other hand, cannot do so because each process has a copy of the program's memory space.

- Copying memory for a new process adds an additional performance overhead relative to creating a new thread. However, the copy is performed only when the memory is changed, so the penalty is minimal if the child process only reads memory.

- Threads should be used for programs that need fine-grained parallelism. For example, if a problem can be broken into multiple, nearly identical tasks, threads may be a good choice. Processes should be used for programs that need coarser parallelism.

- Sharing data among threads is trivial because threads share the same memory. (However, great care must be taken to avoid race conditions, as described previously.) Sharing data among processes requires the use of IPC mechanisms. This can be more cumbersome but makes multiple processes less likely to suffer from concurrency bugs.

1 Interprocess Communication

- The exit status of a child process. That is the simplest form of communication between two processes.
- Interprocess communication (IPC) is the transfer of data among processes.
- We discuss five types of interprocess communication:
  1. *Shared memory* permits processes to communicate by simply reading and writing to a specified memory location.
  2. *Mapped memory* is similar to shared memory, except that it is associated with a file in the filesystem.
  3. *Pipe*s permit sequential communication from one process to a related process.
  4. *FIFO*s are similar to pipes, except that unrelated processes can communicate because the pipe is given a name in the filesystem.
  5. *Socket*s support communication between unrelated processes even on different computers.
- These types of IPC differ by the following criteria:
  - Whether they restrict communication
    - to related processes,
    - to unrelated processes sharing the same filesystem,
    - to any computer connected to a network.
  - Whether a communicating process is limited
    - to only write data,
    - to only read data.
  - The number of processes permitted to communicate.
  - Whether the communicating processes are synchronized by the IPC for example, a reading process halts until data is available to read.

1.1 Shared Memory

Shared memory allows two or more processes to access the same memory as if they all called **malloc** and were returned pointers to the same actual memory. When one process changes the memory, all the other processes see the modification.

1.1.1 Fast Local Communication

- Access to the shared memory is as fast as accessing a process's non-shared memory, and it does not require a system call or entry to the kernel. It also avoids copying data unnecessarily.
- The kernel does not synchronize accesses to shared memory, you must provide your own synchronization.
- A common strategy to avoid these race conditions is to use semaphores.

1.1.2 The Memory Model

- Allocating a new shared memory segment causes virtual memory pages to be created.
- To permit a process to use the shared memory segment, a process attaches it, which adds entries mapping from its virtual memory to the segment's shared pages.
- When finished with the segment, these mapping entries are removed. When no more processes want to access these shared memory segments, exactly one process must deallocate the virtual memory pages.
- All shared memory segments are allocated as integral multiples of the system's **page size**, which is the number of bytes in a page of memory. On Linux systems, the page size is 4KB, but you should obtain this value by calling the **getpagesize** function.

1.1.3 Allocation

- A process allocates a shared memory segment using **shmget** (SHared Memory GET).
  - First parameter is an integer *key* that specifies which segment to create. Unrelated processes can access the same shared segment by specifying the same *key value*. Unfortunately, other processes may have also chosen the same fixed key, which could lead to conflict. Using the special constant **IPC_PRIVATE** as the key value guarantees that a brand new memory segment is created.
  - Second parameter specifies the number of bytes in the segment. Because segments are allocated using pages, the number of actually allocated bytes is rounded up to an integral multiple of the page size.

- Third parameter is the bitwise or flag values that specify options to **shmget**. The flag values include these:
  - **IPC_CREAT**: This flag indicates that a new segment should be created.This permits creating a new segment while specifying a key value.
  - **IPC_EXCL**: This flag, which is always used with **IPC_CREAT**, causes **shmget** to fail if a segment key is specified that already exists. If this flag is not given and the key of an existing segment is used, **shmget** returns the existing segment instead of creating a new one.
  - Mode flags; **S_IRUSR** and **S_IWUSR** specify read and write permissions for the owner of the shared memory segment, and **S_IROTH** and **S_IWOTH** specify read and write permissions for others.

```
int segment_id=shmget(shm_key,getpagesize(),
                      IPC_CREAT | S_IRUSR | S_IWUSER);
```

- If the call succeeds, **shmget** returns a segment identifier. If the shared memory segment already exists, the access permissions are verified and a check is made to ensure that the segment is not marked for destruction.

1.1.4 Attachment and Detachment

- To make the shared memory segment available, a process must use **shmat**, (SHared Memory ATtach).
  - Pass the shared memory segment identifier **SHMID** returned by **shmget**.
  - The second argument is a pointer that specifies where in your process's address space you want to map the shared memory; if you specify NULL, Linux will choose an available address.
  - The third argument is a flag, which can include the following:
    - **SHM_RND** indicates that the address specified for the second parameter should be rounded down to a multiple of the page size.
    - **SHM_RDONLY** indicates that the segment will be only read, not written.

- If the call succeeds, it returns the address of the attached shared segment. Children created by calls to **fork** inherit attached shared segments; they can detach the shared memory segments, if desired.

1.1.5 Controlling and Deallocating Shared Memory

- The **shmctl** (SHared Memory ConTroL) call returns information about a shared memory segment and can modify it.
  - The first parameter is a shared memory segment identifier.
  - To obtain information about a shared memory segment, pass **IPC_STAT** as the second argument and a pointer to a struct **shmid_ds**.
  - To remove a segment, pass **IPC_RMID** as the second argument, and pass NULL as the third argument. The segment is removed when the last process that has attached it finally detaches it.
  - Each shared memory segment should be explicitly deallocated using **shmctl** when you are finished with it, to avoid violating the system wide limit on the total number of shared memory segments. Invoking **exit** and **exec** detaches memory segments but does not deallocate them.
  - The following program (see Fig. 1.1.5) illustrates the use of shared memory.

```
$ ipcs -m
$ ipcrm shm 1627649
```

1.2 Processes Semaphores

Process semaphores are allocated, used, and deallocated like shared memory segments. Although a single semaphore is sufficient for almost all uses, process semaphores come in sets.

1.2.1 Allocation and Deallocation

- The calls **semget** and **semctl** allocate and deallocate semaphores, which is analogous to **shmget** and **shmctl** for shared memory. Invoke **semget** with:
  - a key specifying a semaphore set,

```
#include <stdio.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main ()
{
  int segment_id;
  char* shared_memory;
  struct shmid_ds shmbuffer;
  int segment_size;
  const int shared_segment_size = 0x6400;
  /* Allocate a shared memory segment.  */
  segment_id = shmget (IPC_PRIVATE, shared_segment_size,
       IPC_CREAT | IPC_EXCL | S_IRUSR | S_IWUSR);
  /* Attach the shared memory segment.  */
  shared_memory = (char*) shmat (segment_id, 0, 0);
  printf ("shared memory attached at address %p\n", shared_memory);
  /* Determine the segment's size.  */
  shmctl (segment_id, IPC_STAT, &shmbuffer);
  segment_size = shmbuffer.shm_segsz;
  printf ("segment size: %d\n", segment_size);
  /* Write a string to the shared memory segment.  */
  sprintf (shared_memory, "Hello, world.");
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);
/* Reattach the shared memory segment, at a different address.  */
shared_memory = (char*) shmat (segment_id, (void*) 0x5000000, 0);
  printf ("shared memory reattached at address %p\n", shared_memory);
  /* Print out the string from shared memory.  */
  printf ("%s\n", shared_memory);
  /* Detach the shared memory segment.  */
  shmdt (shared_memory);
  /* Deallocate the shared memory segment.  */
  shmctl (segment_id, IPC_RMID, 0);
  return 0;
}
```

Figure 3: Exercise Shared Memory

  - the number of semaphores in the set,
  - permission flags as for **shmget**

- The return value is a semaphore set identifier. You can obtain the identifier of an existing semaphore set by specifying the right key value.
- Semaphores continue to exist even after all processes using them have terminated. The last process to use a semaphore set must explicitly remove it. To do so, invoke **semctl** with
  - the semaphore identifier,
  - the number of semaphores in the set,
  - **IPC_RMID** as the third argument,
  - any *union semun* value as the fourth argument (which is ignored).
- Unlike shared memory segments, removing a semaphore set causes Linux to deallocate immediately.
- The following code (see Fig. 1.2.1) presents functions to allocate and deallocate a binary semaphore.

```
#include <sys/ipc.h>
#include <sys/sem.h>
#include <sys/types.h>
/* We must define union semun ourselves.  */
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
/* Obtain a binary semaphore's ID, allocating if necessary.  */
int binary_semaphore_allocation (key_t key, int sem_flags)
{
  return semget (key, 1, sem_flags);
}
/* Deallocate a binary semaphore.  All users must have finished
their use. Returns -1 on failure.  */
int binary_semaphore_deallocate (int semid)
{
  union semun ignored_argument;
  return semctl (semid, 1, IPC_RMID, ignored_argument);
}
```

Figure 4: Allocating and Deallocating a Binary Semaphore

1.2.2 Initializing Semaphores

- Allocating and initializing semaphores are two separate operations.
  - To initialize a semaphore, use **semctl** with zero as the second argument,
  - **SETALL** as the third argument,
  - For the fourth argument, you must create a *union semun object* and point its array field at an **array** of unsigned short values. Each value is used to initialize one semaphore in the set.
- The following code (see Fig. 1.2.2) presents a function that initializes a binary semaphore.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/* We must define union semun ourselves.  */
union semun {
  int val;
  struct semid_ds *buf;
  unsigned short int *array;
  struct seminfo *__buf;
};
/* Initialize a binary semaphore with a value of one.  */
int binary_semaphore_initialize (int semid)
{
  union semun argument;
  unsigned short values[1];
  values[0] = 1;
  argument.array = values;
  return semctl (semid, 0, SETALL, argument);
}
```

Figure 5: Initializing a Binary Semaphore

1.2.3 Wait and Post Operations

- Each semaphore has a non-negative value and supports *wait* and *post* operations. The **semop** system call implements both operations.
  - First parameter specifies a semaphore set identifier,

  - Second parameter is an array of **struct sembuf** elements, which specify the operations you want to perform. The fields of struct sembuf are listed:
    * **sem_num** is the semaphore number in the semaphore set on which the operation is performed.
    * **sem_op** is an integer that specifies the semaphore operation.
    * **sem_flg** is a flag value. Specify **IPC_NOWAIT** to prevent the operation from blocking; if the operation would have blocked, the call to **semop** fails instead. If you specify **SEM_UNDO**, Linux automatically undoes the operation on the semaphore when the process exits.
  - The third parameter is the length of this array.

- The following code (see Fig. 1.2.3) illustrates wait and post operations for a binary semaphore.

```
$ ipcs -s
$ ipcrm sem 5790517
```

1.3 Mapped Memory

- Mapped memory permits different processes to communicate via a shared file.
- Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.
- Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space.

1.3.1 Mapping an Ordinary File

- To map an ordinary file to a process's memory, use the **mmap** (Memory MAPped) call.
  - The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address.
  - The second argument is the length of the map in bytes.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/* Wait on a binary semaphore.  Block until the semaphore value
is positive, then decrement it by one.  */
int binary_semaphore_wait (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Decrement by 1.  */
  operations[0].sem_op = -1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;
  return semop (semid, operations, 1);
}
/* Post to a binary semaphore: increment its value by one.  This
   returns immediately.  */
int binary_semaphore_post (int semid)
{
  struct sembuf operations[1];
  /* Use the first (and only) semaphore.  */
  operations[0].sem_num = 0;
  /* Increment by 1.  */
  operations[0].sem_op = 1;
  /* Permit undo'ing.  */
  operations[0].sem_flg = SEM_UNDO;
  return semop (semid, operations, 1);
}
```

Figure 6: Wait and Post Operations for a Binary Semaphore

  - The third argument specifies the protection on the mapped address range. The protection consists of a bitwise or of **PROT_READ**, **PROT_WRITE**, and **PROT_EXEC**.
  - The fourth argument is a flag value that specifies additional options. The flag value is a bitwise or of these constraints:
    * **MAP_PRIVATE** Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with **MAP_SHARED**.
    * **MAP_SHARED** Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC. This mode may not be used with **MAP_PRIVATE**.
  - The fifth argument is a file descriptor opened to the file to be mapped.
  - The last argument is the offset from the beginning of the file from which to start the map.

- When you are finished with a memory mapping, release it by using **munmap**. Pass it the start address and length of the mapped memory region. Linux automatically unmaps mapped regions when a process terminates.

- The following code (see Fig. 1.3.1) generates a random number and writes it to a memory-mapped file.

- The following code (see Fig. 1.3.1) reads the number, prints it, and replaces it in the memory-mapped file with double the value.

```
$ ./mmap-write /tmp/integer-file
$ cat /tmp/integer-file
42
$ ./mmap-read /tmp/integer-file
value: 42
$ cat /tmp/integer-file
84
```

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
/* Return a uniformly random number in the range [low,high].  */
int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}
int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  /* Seed the random number generator.  */
  srand (time (NULL));
  /* Prepare a file large enough to hold an unsigned integer.  */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);
  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
  close (fd);
  /* Write a random integer to memory-mapped area.  */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);
  return 0;
}
```

Figure 7: Write a Random Number to a Memory-Mapped File

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100
int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;
  int integer;
  /* Open the file.  */
  fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
  /* Create the memory-mapping.  */
  file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
      MAP_SHARED, fd, 0);
  close (fd);
  /* Read the integer, print it out, and double it.  */
  sscanf (file_memory, "%d", &integer);
  printf ("value: %d\n", integer);
  sprintf ((char*) file_memory, "%d\n", 2 * integer);
  /* Release the memory (unnecessary since the program exits).  */
  munmap (file_memory, FILE_LENGTH);
  return 0;
}
```

Figure 8: Read an Integer from a Memory-Mapped File, and Double It

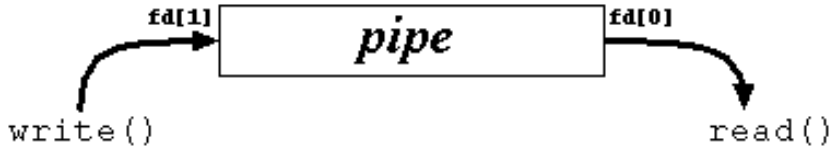

Figure 9: How a pipe is organized.

1.4 Pipes

- A **pipe** is a communication device that permits unidirectional communication. (see Fig. 1.4)
  - data written to the *write end* of the pipe,
  - read back from the *read end*,
  - pipes are serial devices; the data is always read from the pipe in the same order it was written.
- This shell command causes the shell to produce two child processes, one for **ls** and one for **less**:

```
$ ls | less
```

- A pipe's data capacity is limited.
  - if the writer process writes faster than the reader process consumes the data,
  - if the pipe cannot store more data, the writer process blocks until more capacity becomes available,
  - if the reader tries to read but no data is available, it blocks until data becomes available.
- Thus, the pipe *automatically synchronizes* the two processes.
- To create a pipe, invoke the **pipe** command. Supply an integer array of size 2. The call to pipe stores the reading file descriptor in array position 0 and the writing file descriptor in position 1.

```
int pipe_fds[2];
int read_fd;
int write_fd;
```

```
pipe (pipe_fds);
read_fd = pipe_fds[0];
write_fd = pipe_fds[1];
```

1.4.1 Communication Between Parent and Child Processes

- A call to pipe creates file descriptors, which are valid only within that process and its children.
- A process's file descriptors cannot be passed to unrelated processes; however, when the process calls **fork**, file descriptors are copied to the new child process. Thus, pipes can connect only *related processes*.
- In the following program, (see Fig. 1.4.1) a **fork** spawns a child process. The child inherits the pipe file descriptors. The parent writes a string to the pipe, and the child reads it out.

1.4.2 Redirecting the Standard Input, Output, and Error Streams

- Frequently, you will want to create a child process and set up one end of a pipe as its standard input or standard output.
- Using the **dup2** call, you can equate one file descriptor with another.

  ```
  dup2(fd,STDIN_FILENO);
  ```

- The symbolic constant **STDIN_FILENO** represents the file descriptor for the standard input, which has the value 0.
- The following program, (see Fig. 1.4.2) uses **dup2** to send the output from a **pipe** to the **sort** command.

```
#include <stdlib.h>
#include <stdio.h>
#include <unistd.h>
/* Write COUNT copies of MESSAGE to STREAM, pausing for a second
   between each.  */
void writer (const char* message, int count, FILE* stream)
{
  for (; count > 0; --count) {
/* Write the message to the stream, and send it off immediately.  */
    fprintf (stream, "%s\n", message);
    fflush (stream);
    /* Snooze a while.  */
    sleep (1);
  }
}
/* Read random strings from the stream as long as possible.  */
void reader (FILE* stream)
{
  char buffer[1024];
  /* Read until we hit the end of the stream.  fgets reads until
     either a newline or the end-of-file.  */
  while (!feof (stream)
 && !ferror (stream)
 && fgets (buffer, sizeof (buffer), stream) != NULL)
    fputs (buffer, stdout);
}
int main ()
{
  int fds[2];
  pid_t pid;
  /* Create a pipe.  File descriptors for the two ends of the pipe are placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
    FILE* stream;
/* This is the child process.  Close our copy of the write end of the file descriptor.  */
    close (fds[1]);
/* Convert the read file descriptor to a FILE object, and read  from it.  */
    stream = fdopen (fds[0], "r");
    reader (stream);
    close (fds[0]);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
    /* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
/* Convert the write file descriptor to a FILE object, and write
       to it.  */
    stream = fdopen (fds[1], "w");
    writer ("Hello, world.", 5, stream);
    close (fds[1]);
  }
  return 0;
}
```

Figure 10: Using a Pipe to Communicate with a Child Process

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <unistd.h>
int main ()
{
  int fds[2];
  pid_t pid;
/* Create a pipe.  File descriptors for the two ends of the pipe
are placed in fds.  */
  pipe (fds);
  /* Fork a child process.  */
  pid = fork ();
  if (pid == (pid_t) 0) {
/* This is the child process.  Close our copy of the write end of
the file descriptor.  */
    close (fds[1]);
/* Connect the read end of the pipe to standard input.  */
    dup2 (fds[0], STDIN_FILENO);
    /* Replace the child process with the "sort" program.  */
    execlp ("sort", "sort", 0);
  }
  else {
    /* This is the parent process.  */
    FILE* stream;
/* Close our copy of the read end of the file descriptor.  */
    close (fds[0]);
/* Convert the write file descriptor to a FILE object, and write
to it.  */
    stream = fdopen (fds[1], "w");
    fprintf (stream, "This is a test.\n");
    fprintf (stream, "Hello, world.\n");
    fprintf (stream, "My dog has fleas.\n");
    fprintf (stream, "This program is great.\n");
    fprintf (stream, "One fish, two fish.\n");
    fflush (stream);
    close (fds[1]);
/* Wait for the child process to finish.  */
    waitpid (pid, NULL, 0);
  }
  return 0;
}
```

Figure 11: Redirect Output from a Pipe with dup2

1.5 FIFOs

- A first-in, first-out (FIFO) file is a *pipe* that has a name in the filesystem.
- Any process can open or close the FIFO; the processes on either end of the pipe need not be related to each other.
- FIFOs are also called **named pipes**.

```
$ mkfifo /tmp/fifo
$ ls -l /tmp/fifo
```

- In one window, read from the FIFO by invoking the following:

```
$ cat < /tmp/fifo
```

- In a second window, write to the FIFO by invoking this:

```
$ cat > /tmp/fifo
```

- Then type in some lines of text. Each time you press *Enter*, the line of text is sent through the FIFO and appears in the first window. Close the FIFO by pressing Ctrl+D in the second window.
- Remove the FIFO with this line:

```
$ rm /tmp/fifo
```