Operating Systems
Lecture No. 6

Reading Material

- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for the `fork()` system call

Summary

- Process creation and termination
- Process management in UNIX/Linux— system calls: `fork`, `exec`, `wait`, `exit`
- Sample codes

Operations on Processes

The processes in the system execute concurrently and they must be created and deleted dynamically thus the operating system must provide the mechanism for the creation and deletion of processes.

Process Creation

A process may create several new processes via a create-process system call during the course of its execution. The creating process is called a **parent process** while the new processes are called the **children** of that process. Each of these new processes may in turn create other processes, forming a tree of processes. Figure 6.1 shows partially the process tree in a UNIX/Linux system.

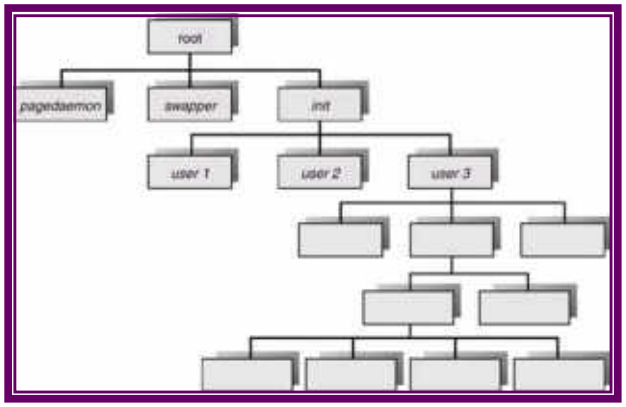

Figure 6.1 Process tree in UNIX/Linux

In general, a process will need certain resources (such as CPU time, memory files, I/O devices) to accomplish its task. When a process creates a sub process, also known as a child, that sub process may be able to obtain its resources directly from the operating system or may be constrained to a subset of the resources of the parent process. The parent may have to partition its resources among several of its children. Restricting a

process to a subset of the parent's resources prevents a process from overloading the system by creating too many sub processes.

When a process is created it obtains in addition to various physical and logical resources, initialization data that may be passed along from the parent process to the child process. When a process creates a new process, two possibilities exist in terms of execution:

1. The parent continues to execute concurrently with its children.
2. The parent waits until some or all of its children have terminated.

There are also two possibilities in terms of the address space of the new process:

1. The child process is a duplicate of the parent process.
2. The child process has a program loaded into it.

In order to consider these different implementations let us consider the UNIX operating system. In UNIX its process identifier identifies a process, which is a unique integer. A new process is created by the `fork` system call. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. Both processes continue execution at the instruction after the `fork` call, with one difference, the return code for the `fork` system call is zero for the child process, while the process identifier of the child is returned to the parent process.

Typically the `execlp` system call is used after a `fork` system call by one of the two processes to replace the process' memory space with a new program. The `execlp` system call loads a binary file in memory –destroying the memory image of the program containing the `execlp` system call.—and starts its execution. In this manner, the two processes are able to communicate and then go their separate ways. The parent can then create more children, or if it has nothing else to do while the child runs, it can issue a `wait` system call to move itself off the ready queue until the termination of the child. The parent waits for the child process to terminate, and then it resumes from the call to wait where it completes using the `exit` system call.

Process termination

A process terminates when it finishes executing its final statement and asks the operating system to delete it by calling the exit system call. At that point, the process may return data to its parent process (via the wait system call). All the resources of the process including physical and virtual memory, open the files and I/O buffers – are de allocated by the operating system.

Termination occurs under additional circumstances. A process can cause the termination of another via an appropriate system call (such as abort). Usually only the parent of the process that is to be terminated can invoke this system call. Therefore parents need to know the identities of its children, and thus when one process creates another process, the identity of the newly created process is passed to the parent.

A parent may terminate the execution of one of its children for a variety of reasons, such as:

- The child has exceeded its usage of some of the resources that it has been allocated. This requires the parent to have a mechanism to inspect the state of its children.
- The task assigned to the child is no longer required.

- The parent is exiting, and the operating system does not allow a child to continue if its parent terminates. On such a system, if a process terminates either normally or abnormally, then all its children must also be terminated. This phenomenon referred to as cascading termination, is normally initiated by the operating system.

Considering an example from UNIX, we can terminate a process by using the `exit` system call, its parent process may wait for the termination of a child process by using the wait system call. The wait system call returns the process identifier of a terminated child, so that the parent can tell which of its possibly many children has terminated. If the parent terminates however all its children have assigned as their new parent, the `init` process. Thus the children still have a parent to collect their status and execution statistics.

The `fork()` system call

When the fork system call is executed, a new process is created. The original process is called the parent process whereas the process is called the child process. The new process consists of a copy of the address space of the parent. This mechanism allows the parent process to communicate easily with the child process. On success, both processes continue execution at the instruction after the `fork` call, with one difference, the return code for the `fork` system call is zero for the child process, while the process identifier of the child is returned to the parent process. On failure, a -1 will be returned in the parent's context, no child process will be created, and an error number will be set appropriately.

The synopsis of the fork system call is as follows:

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

```
main()
{
     int pid;
     ...
     pid = fork();
     if (pid == 0) {
     /* Code for child */
     ...
     }
     else {
     /* Code for parent */
     ...
     }
     ...
}
```

Figure 6.2 Sample code showing use of the `fork()` system call

Figure 6.2 shows sample code, showing the use of the `fork()` system call and Figure 6.3 shows the semantics of the fork system call. As shown in Figure 6.3, `fork()`

creates an exact memory image of the parent process and returns 0 to the child process and the process ID of the child process to the parent process.

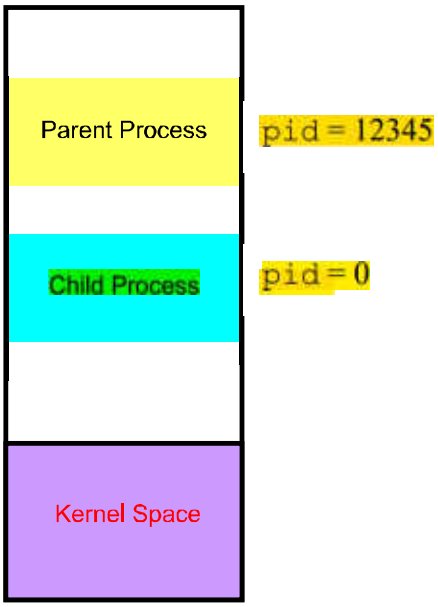

Figure 6.3 Semantics of the fork system call

After the `fork()` system call the parent and the child share the following:

- Environment
- Open file descriptor table
- Signal handling settings
- Nice value
- Current working directory
- Root directory
- File mode creation mask (umask)

The following things are different in the parent and the child:

- Different process ID (PID)
- Different parent process ID (PPID)
- Child has its own copy of parent's file descriptors

The `fork()` system may fail due to a number of reasons. One reason maybe that the maximum number of processes allowed to execute under one user has exceeded, another could be that the maximum number of processes allowed on the system has exceeded. Yet another reason could be that there is not enough swap space.

Operating Systems
Lecture No. 7

Reading Material

- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `execlp()`, `exit()`, and `wait()` system calls

Summary

- The `execlp()`, `wait()`, and `exec()` system calls and sample code
- Cooperating processes
- Producer-consumer problem
- Interprocess communication (IPC) and process synchronization

The `wait()` system call

The wait system call suspends the calling process until one of the immediate children terminate, or until a child that is being traced stops because it has hit an event of interest. The wait will return prematurely if a signal is received. If all child processes stopped or terminated prior to the call on wait, return is immediate. If the call is successful, the process ID of a child is returned. If the parent terminates however all its children have assigned as their new parent, the init process. Thus the children still have a parent to collect their status and execution statistics. The synopsis of the wait system call is as follows:

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait(int *stat_loc);
```

A **zombie process** is a process that has terminated but whose exit status has not yet been received by its parent process or by init. Sample code showing the use of `fork()` and `wait()` system calls is given in Figure 7.1 below.

```
#include <stdio.h>
void main(){
  int pid, status;
  pid = fork();
  if(pid == -1) {
   printf("fork failed\n");
   exit(1);
  }
   if(pid == 0) {   /* Child */
       printf("Child here!\n");
       exit(0);
  }
  else { /* Parent */
    wait(&status);
```

```
    printf("Well done kid!\n");
    exit(0);
  }
}
```

Figure 7.1 Sample code showing use of the `fork()` and `wait()` system calls

The `execlp()` system call

Typically, the `execlp()` system call is used after a `fork()` system call by one of the two processes to replace the process' memory space with a new program. The new process image is constructed from an ordinary, executable file. This file is either an executable object file, or a file of data for an interpreter. There can be no return from a successful *exec* because the calling process image is overlaid by the new process image. In this manner, the two processes are able to communicate and then go their separate ways. The synopsis of the `execlp()` system call is given below:

```
#include <unistd.h>
int execlp (const char *file, const,char *arg0, ...,
            const char *argn,(char *)0);
```

Sample code showing the use of `fork()` and `execlp()` system calls is given in Figure 7.2 below.

```
#include <stdio.h>
void main()
{
  int pid, status;

  pid = fork();
  if(pid == -1) {
    printf("fork failed\n");
    exit(1);
  }
  if(pid == 0) {  /* Child */
      if (execlp("/bin/ls", "ls", NULL)< 0) {
        printf("exec failed\n");
        exit(1);
      }
  }
  else { /* Parent */
    wait(&status);
    printf("Well done kid!\n");
    exit(0);
  }
}
```

Figure 7.2 Sample code showing use of `fork()`, `execlp()`, `wait()`, and `exit()`

The semantics of `fork()`, followed by an execlp() system call are shown In Figure 7.3 below.

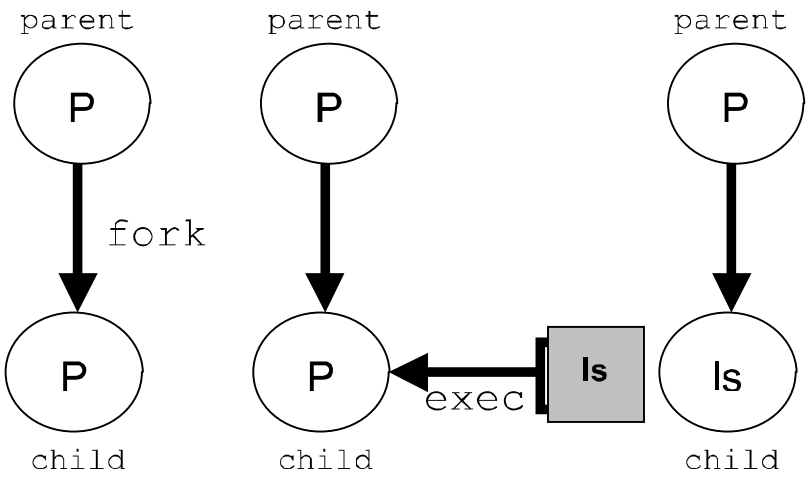

Figure 7.3 Semantics of `fork()` followed by `exec()`

Cooperating Processes

The concurrent processes executing in the operating system may be either independent processes or cooperating processes. A process is independent if it cannot affect or be affected by any other process executing in the system. Clearly any process that shares data with other processes is a cooperating process. The advantages of cooperating processes are:

- **Information sharing:** Since several users may be interested in the same piece of information (for instance, a shared file) we must provide an environment to allow concurrent users to access these types of resources.
- **Computation speedup:** If we want a particular task to run faster, we must break it into subtasks each of which will be running in parallel with the others. Such a speedup can be obtained only if the computer has multiple processing elements (such as CPU's or I/O channels).
- **Modularity:** We may want to construct the system in a modular fashion, dividing the system functions into separate processes or threads.
- **Convenience:** Even an individual user may have many tasks on which to work at one time. For instance, a user may be editing, printing, and compiling in parallel.

To illustrate the concept of communicating processes, let us consider the producer-consumer problem. A **producer** process produces information that is consumed by a **consumer** process. For example, a compiler may produce assembly code that is consumed by an assembler. To allow a producer and consumer to run concurrently, we must have available a buffer of items that can be filled by a producer and emptied by a consumer. The producer and consumer must be synchronized so that the consumer does not try to consume an item that has not yet been produced. The bounded buffer problem assumes a fixed buffer size, and the consumer must wait if the buffer is empty and the producer must wait if the buffer is full, whereas the unbounded buffer places no practical limit on the size of the buffer. Figure 7.4 shows the problem in a diagram. This buffer may be provided by interprocess communication (discussed in the next section) or with the use of shared memory.

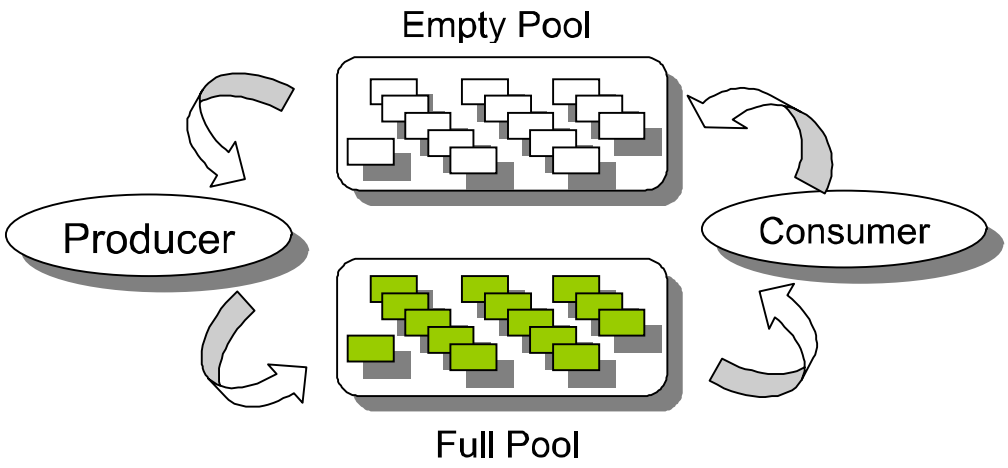

Figure 7.4 The producer-consumer problem

Figure 7.5 shows the shared buffer and other variables used by the producer and consumer processes.

```
#define BUFFER_SIZE 10
typedef struct
{
…
} item;
item buffer[BUFFER_SIZE];
int in=0;
int out=0;
```

Figure 7.5 Shared buffer and variables used by the producer and consumer processes

The shared buffer is implemented as a circular array with two logical pointers: in an out. The 'in' variable points to the next free position in the buffer; 'out' points to the first full position in the buffer. The buffer is empty when `in==out`, the buffer is full when `((in+1)%BUFFER_SIZE)==out`. The code structures for the producer and consumer processes are shown in Figure 7.6.

```
Producer Process
while(1) {
    /*Produce an item in nextProduced*/
    while(((in+1)%BUFFER_SIZE)==out); /*do nothing*/
       buffer[in]=nextProduced;
       in=(in+1)%BUFFER_SIZE;
}
Consumer Process
while(1) {
   while(in == out); //do nothing
   nextConsumed=buffer[out];
   out=(out+1)%BUFFER_SIZE;
   /*Consume the item in nextConsumed*/
}
```

Figure 7.6 Code structures for the producer and consumer processes

Operating Systems
Lecture No. 8

Reading Material

- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `pipe()`, `fork()`, `read()`, `write()`, `close()`, and `wait()` system calls

Summary

- Interprocess communication (IPC) and process synchronization
- UNIX/Linux IPC tools (pipe, named pipe—FIFO, socket, TLI, message queue, shared memory)
- Use of UNIC/Linux pipe in a sample program

Interprocess Communication (IPC)

IPC provides a mechanism to allow processes to communicate and to synchronize their actions without sharing the same address space. We discuss in this section the various message passing techniques and issues related to them.

Message Passing System

The function of a message system is to allow processes to communicate without the need to resort to the shared data. Messages sent by a process may be of either fixed or variable size. If processes P and Q want to communicate, a communication link must exist between them and they must send messages to and receive messages from each other through this link. Here are several methods for logically implementing a link and the send and receive options:

- Direct or indirect communication
- Symmetric or asymmetric communication
- Automatic or explicit buffering
- Send by copy or send by reference
- Fixed size or variable size messages

We now look at the different types of message systems used for IPC.

Direct Communication

With direct communication, each process that wants to communicate must explicitly name the recipient or sender of the communication. The send and receive primitives are defined as:

- Send(P, message) – send a message to process P
- Receive(Q, message) – receive a message from process Q.

A communication link in this scheme has the following properties:

- A link is established automatically between every pair of processes that want to communicate. The processes need to know only each other's identity to communicate
- A link is associated with exactly two processes.
- Exactly one link exists between each pair of processes.

Unlike this symmetric addressing scheme, a variant of this scheme employs asymmetric addressing, in which the recipient is not required to name the sender.

- Send(P, message) – send a message to process P
- Receive(id, message) – receive a message from any process; the variable id is set to the name of the process with which communication has taken place.

Indirect Communication

With indirect communication, messages can be sent to and received from mailboxes. Here, two processes can communicate only if they share a mailbox. The send and receive primitives are defined as:

- Send(A, message) – send a message to mailbox A.
- Receive(A, message) – receive a message from mailbox A.

A communication link in this scheme has the following properties:

- A link is established between a pair of processes only if both members have a shared mailbox.
- A link is associated with more than two processes.
- A number of different links may exist between each pair of communicating processes, with each link corresponding to one mailbox.

Synchronization

Communication between processes takes place by calls to send and receive primitives (i.e., functions). Message passing may be either blocking or non-blocking also called as synchronous and asynchronous.

- **Blocking send:** The sending process is blocked until the receiving process or the mailbox receives the message.
- **Non-blocking send:** The sending process sends the message and resumes operation.
- **Blocking receive:** The receiver blocks until a message is available.
- **Non-blocking receiver:** The receiver receives either a valid message or a null.

Buffering

Whether the communication is direct or indirect, messages exchanged by the processes reside in a temporary queue. This queue can be implemented in three ways:

- **Zero Capacity:** The queue has maximum length zero, thus the link cannot have any messages waiting in it. In this case the sender must block until the message has been received.
- **Bounded Capacity:** This queue has finite length n; thus at most n messages can reside in it. If the queue is not full when a new message is sent, the later is placed in the queue and the sender resumes operation. If the queue is full, the sender blocks until space is available.

- **Unbounded Capacity:** The queue has infinite length; thus the sender never blocks.

UNIX/Linux IPC Tools

UNIX and Linux operating systems provide many tools for interprocess communication, mostly in the form of APIs but some also for use at the command line. Here are some of the commonly supported IPC tools in the two operating systems.

- Pipe
- Named pipe (FIFO)
- BSD Socket
- TLI
- Message queue
- Shared memory
- Etc.

Overview of `read()`, `write()`, and `close()` System Calls

We need to understand the purpose and syntax of the read, write and close system calls so that we may move on to understand how communication works between various Linux processes. The read system call is used to read data from a file descriptor. The synopsis of this system call is:

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

`read()` attempts to read up to count bytes from file descriptor fd into the buffer starting at buf. If count is zero, `read()` returns zero and has no other results. If count is greater than SSIZE_MAX, the result is unspecified. On success, `read()` returns the number of bytes read (zero indicates end of file) and advances the file position pointer by this number.

The `write()` system call is used to write to a file. Its synopsis is as follows:

```
#include <unistd.h>
ssize_t write(int fd, const void *buf, size_t count);
```

`write()` attempts to write up to count bytes to the file referenced by the file descriptor fd from the buffer starting at buf. On success, `write()` returns the number of bytes written are returned (zero indicates nothing was written) and advances the file position pointer by this number. On error, `read()` returns -1, and *`errno`* is set appropriately. If count is zero and the file descriptor refers to a regular file, 0 will be returned without causing any other effect.

The `close()` system call is used to close a file descriptor. Its synopsis is:

```
#include <unistd.h>
int close(int fd);
```

`close()` closes a file descriptor, so that it no longer refers to any file and may be reused. If fd is the last copy of a particular file descriptor the resources associated with it are freed; if the descriptor was the last reference to a file which has been removed using

`unlink(2)` the file is deleted. `close()` returns zero on success, or -1 if an error occurred.

Pipes

A UNIX/Linux pipe can be used for IPC between related processes on a system. Communicating processes typically have sibling or parent-child relationship. At the command line, a pipe can be used to connect the standard output of one process to the standard input of another. Pipes provide a method of one-way communication and for this reason may be called half-duplex pipes.

The `pipe()` system call creates a pipe and returns two file descriptors, one for reading and second for writing, as shown in Figure 8.1. The files associated with these file descriptors are streams and are both opened for reading and writing. Naturally, to use such a channel properly, one needs to form some kind of protocol in which data is sent over the pipe. Also, if we want a two-way communication, we'll need two pipes.

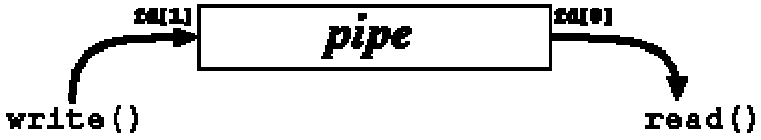

Figure 8.1 A UNIX/Linux pipe with a read end and a write end

The system assures us of one thing: the order in which data is written to the pipe, is the same order as that in which data is read from the pipe. The system also assures that data won't get lost in the middle, unless one of the processes (the sender or the receiver) exits prematurely. The `pipe()` system call is used to create a read-write pipe that may later be used to communicate with a process we'll fork off. The synopsis of the system call is:

```
#include <unistd.h>
int pipe (int fd[2]);
```

Each array element stores a file descriptor. fd[0] is the file descriptor for the read end of the pipe (i.e., the descriptor to be used with the read system call), whereas fd[1] is the file descriptor for the write end of the pipe. (i.e., the descriptor to be used with the write system call).The function returns -1 if the call fails. A pipe is a bounded buffer and the maximum data written is PIPE_BUF, defined in <sys/param.h> in UNIX and in <linux/param.h> in Linux as 5120 and 4096, respectively.

Lets see an example of a two-process system in which the parent process creates a pipe and forks a child process. The child process writes the 'Hello, world!' message to the pipe. The parent process reads this messages and displays it on the monitor screen. Figure 8.2 shows the protocol for this communication and Figure 8.3 shows the corresponding C source code.

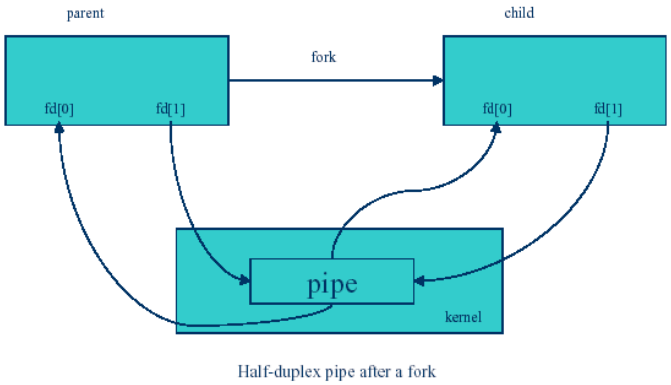

Figure 8.2 Use of UNIX/Linux pipe by parent and child for half-duplex communication

```
/* Parent creates pipe, forks a child, child writes into
   pipe, and parent reads from pipe */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        int pipefd[2], pid, n, rc, nr, status;
        char *testString = "Hello, world!\n", buf[1024];

        rc = pipe (pipefd);
        if (rc < 0) {
                perror("pipe");
                exit(1);
        }
        pid = fork ();
        if (pid < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {  /* Child's Code */
            close(pipefd[0]);
            write(pipefd[1], testString, strlen(testString));
            close(pipefd[1]);
            exit(0);
        }
        /* Parent's Code */
        close(pipefd[1]);
        n = strlen(testString);
        nr = read(pipefd[0], buf, nA);
        rc = write(1, buf, nr);
        wait(&status);
        printf("Good work child!\n");
        return(0);
}
```

Figure 8.3 Sample code showing use of UNIX/Linux pipe for IPC between related processes—child write the "Hello, world!" message to the parent, who reads its and displays it on the monitor screen

In the given program, the parent process first creates a pipe and then forks a child process. On successful execution, the `pipe()` system call creates a pipe, with its read end descriptor stored in pipefd[0] and write end descriptor stored in pipefd[1]. We call `fork()` to create a child process, and then use the fact that the memory image of the child process is identical to the memory image of the parent process, so the pipefd[] array is still defined the same way in both of them, and thus they both have the file descriptors of the pipe. Further more, since the file descriptor table is also copied during the fork, the file descriptors are still valid inside the child process. Thus, the parent and child processes can use the pipe for one-way communication as outlined above.

Operating Systems
Lecture No. 9

Reading Material

- Operating Systems Concepts, Chapter 4
- UNIX/Linux manual pages for `pipe()`, `fork()`, `read()`, `write()`, `close()`, and `wait()` system calls
- Lecture 9 on Virtual TV

Summary

- UNIX/Linux interprocess communication (IPC) tools and associated system calls
- UNIX/Linux standard files and kernel's mechanism for file access
- Use of pipe in a program and at the command line

Unix/Linux IPC Tools

The UNIX and Linux operating systems provide many tools for interprocess communication (IPC). The three most commonly used tools are:

- **Pipe:** Pipes are used for communication between related processes on a system, as shown in Figure 9.1. The communicating processes are typically related by sibling or parent-child relationship.

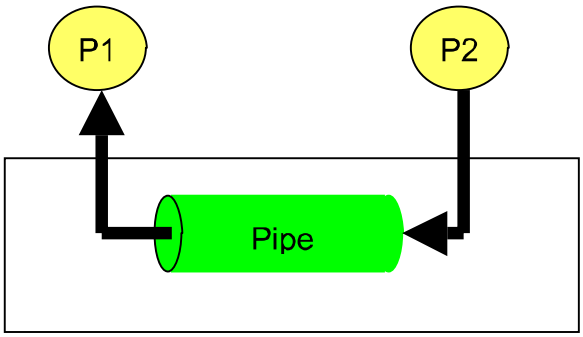

Figure 9.1 Pipes on a UNIX/Linux system

- **Named pipe (FIFO)**: FIFOs (also known as named pipes) are used for communication between related or unrelated processes on a UNIX/Linux system, as shown in Figure 9.2.

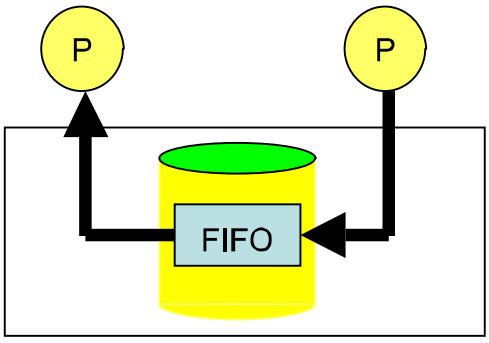

Figure 9.2 Pipes on a UNIX/Linux system

- **BSD Socket**: The BSD sockets are used for communication between related or unrelated processes on the same system or unrelated processes on different systems, as shown in Figure 9.3.

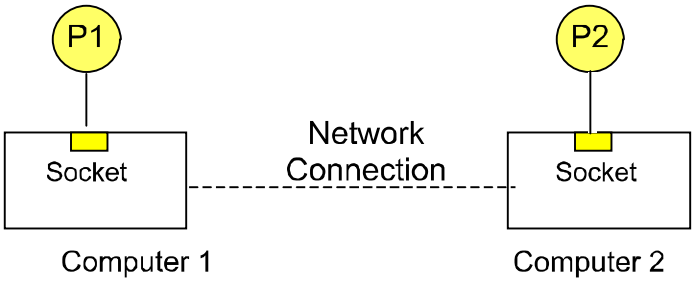

Figure 9.3 Sockets used for IPC between processes on different UNIX/Linux systems

The `open()` System call

The `open()` system call is used to open or create a file. Its synopsis is as follows:

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *pathname, int flags);
int open(const char pathname, int oflag, /* mode_t mode */);
```

The call converts a pathname into a file descriptor (a small, non-negative integer for use in subsequent I/O as with read, write, etc.). When the call is successful, the file descriptor returned will be the lowest file descriptor not currently open for the process. This system call can also specify whether read or write will be blocking or non-blocking.

The 'oflag' argument specifies the purpose of opening the file and 'mode' specifies permission on the file if it is to be created. 'oflag' value is constructed by ORing various flags: O_RDONLY, O_WRONLY, O_RDWR, O_NDELAY (or O_NONBLOCK), O_APPEND, O_CREAT, etc.

`The open()` system call can fail for many reasons, some of which are:

- Non-existent file
- Operation specified is not allowed due to file permissions

- Search not allowed on a component of pathname
- User's disk quota on the file system has been exhausted

The file descriptor returned by the `open()` system call is used in the `read()` and `write()` calls for file (or pipe) I/O.

The `read()` system call

We discussed the `read()` system call in the notes for lecture 8. The call may fail for various reasons, including the following:

- Invalid 'fildes', 'buf', or 'nbyte'
- Signal caught during read

The `write()` system call

The call may fail for various reasons, including the following:

- Invalid argument
- File size limit for process or for system would exceed
- Disk is full

The `close()` system call

As discussed in the notes for lecture 8, the close() system call is used to close a file descriptor. It takes a file (or pipe) descriptor as an argument and closes the corresponding file (or pipe end).

Kernel Mapping of File Descriptors

Figure 9.4 shows the kernel mapping of a file descriptor to the corresponding file. The system-wide File Table contains entries for all of the open files on the system. UNIX/Linux allocates an inode to every (unique) file on the system to store most of the attributes, including file's location. On a read or write call, kernel traverses this mapping to reach the corresponding file.

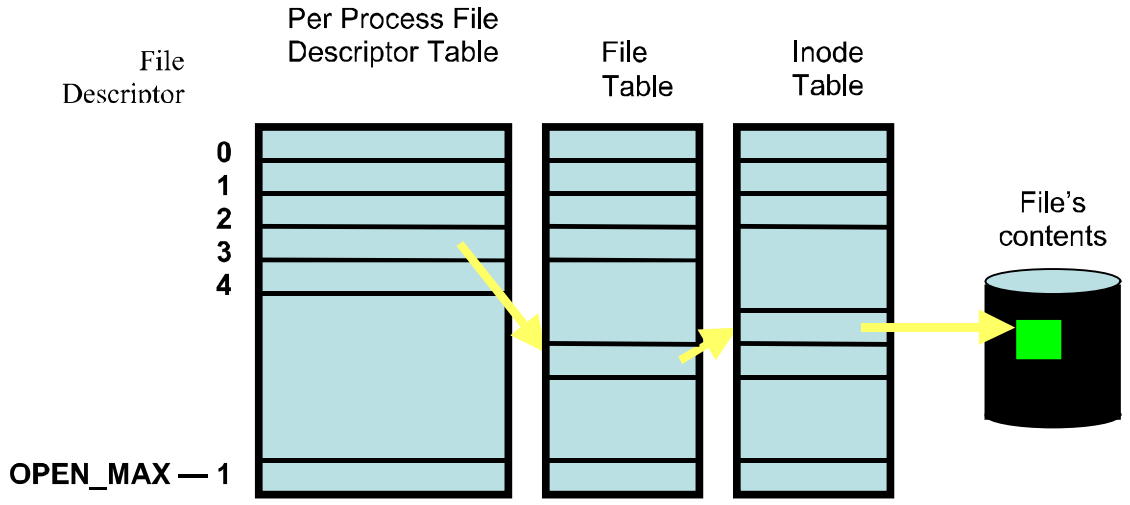

Figure 9.4 File descriptors and their mapping to files

Standard Descriptors in Unix/Linux

Three files are automatically opened by the kernel for every process for the process to read its input from and send its output and error messages to. These files are called **standard files**: standard input, standard output, and standard error. By default, standar d files are attached to the terminal on which a process runs. The descriptors for standard files are known as **standard file descriptors**. Standard files, their descriptors, and their default attachments are:

- Standard input: 0 (keyboard)
- Standard output: 1 (display screen)
- Standard error: 2 (display screen)

The `pipe()` System Call

We discussed the `pipe()` system call in the notes for lecture 8. The `pipe()` system call fails for many reasons, including the following:

- At least two slots are not empty in the PPFDT—too many files or pipes are open in the process
- Buffer space not available in the kernel
- File table is full

Sample Code for IPC with a UNIX/Linux Pipe

We discussed in the notes for lecture 8 a simple protocol for communication between a parent and its child process using a pipe. Figure 9.5 shows the protocol. Code is reproduced in Figure 9.6.

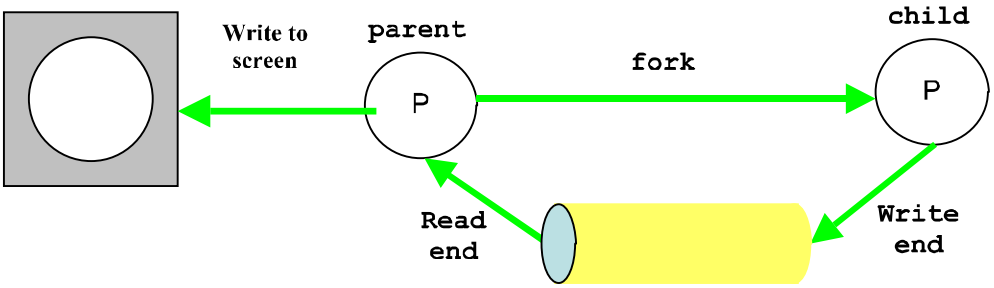

Figure 9.5 IPC between parent and child processes with a UNIX/Linux pipe

```
/* Parent creates pipe, forks a child, child writes into
   pipe, and parent reads from pipe */
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
main()
{
        int pipefd[2], pid, n, rc, nr, status;
        char *testString = "Hello, world!\n", buf[1024];

        rc = pipe (pipefd);
        if (rc < 0) {
                perror("pipe");
```

```
                exit(1);
        }
        pid = fork ();
        if (pid < 0) {
                perror("fork");
                exit(1);
        }
        if (pid == 0) {  /* Child's Code */
            close(pipefd[0]);
            write(pipefd[1], testString, strlen(testString));
            close(pipefd[1]);
            exit(0);
        }
        /* Parent's Code */
        close(pipefd[1]);
        n = strlen(testString);
        nr = read(pipefd[0], buf, nA);
        rc = write(1, buf, nr);
        wait(&status);
        printf("Good work child!\n");
        return(0);
}
```

Figure 9.6 Sample code showing use of UNIX/Linux pipe for IPC between related processes—child write the "Hello, world!" message to the parent, who reads its and displays it on the monitor screen

Command Line Use of UNIX/Linux Pipes

Pipes can also be used on the command line to connect the standard input of one process to the standard input of another. This is done by using the pipe operator which is | and the syntax is as follows:

```
cmd1 | cmd2 | ... | cmdN
```

The semantics of this command line are shown in Figure 9.7.

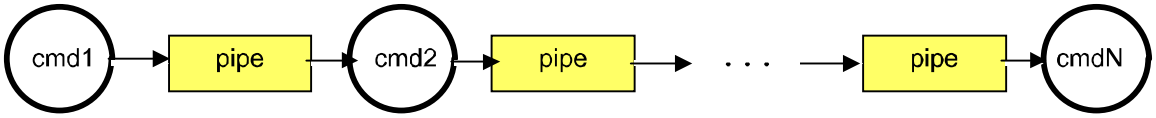

Figure 9.7 Semantics of the command line that connects `cmd1` through `cmdN` via pipes.

The following example shows the use of the pipe operator in a shell command.

```
cat /etc/passwd | grep zaheer
```

The effect of this command is that `grep` command displays lines in the /etc/passwd file that contain the string "zaheer". Figure 9.8 illustrates the semantics of this command.

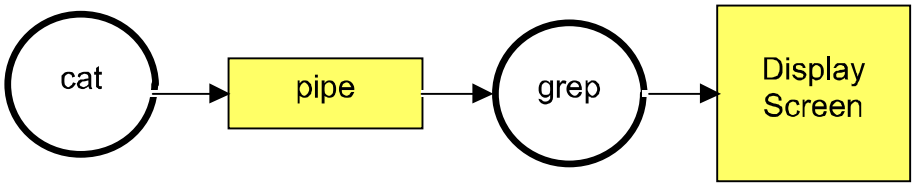

Figure 9.8 Semantics of the cat /etc/passwd | grep zaheer command

The work performed by the above command can be performed by the following sequence of commands without using the pipe operator. The first command saves the /etc/passwd file in the temp1 file and the second command displays those lines in temp1 which contain the string "zaheer". After the temp1 file has been used for the desired work, it is deleted.

```
$ cat /etc/passwd > temp1
$ grep "zaheer" temp1
$ rm temp1
```

Operating Systems
Lecture No. 10

Reading Material

- UNIX/Linux manual pages for the `mknod()` system call, the `mkfifo()` library call, and the `mkfifo` command
- Lecture 10 on Virtual TV

Summary

- Input, output, and error redirection in UNIX/Linux
- FIFOs in UNIX/Linux
- Use of FIFOs in a program and at the command line

Input, output and error redirection in UNIX/Linux

Linux redirection features can be used to detach the default files from stdin, stdout, and stderr and attach other files with them for a single execution of a command. The act of detaching defaults files from stdin, stdout, and stderr and attaching other files with them is known as input, output, and error redirection. In this section, we show the syntax, semantics, and examples of I/O and error redirection.

Input Redirection: Here is the syntax for input redirection:

```
command < input-file
```

or

```
command 0< input-file
```

With this command syntax, keyboard is detached from stdin of 'command' and 'input-file' is attached to it, i.e., 'command' reads input from 'input-file' and not keyboard. Note that 0< operator cannot be used with the C and TC shells. Here is an example use of input redirection. In these examples, the `cat` and `grep` commands read input from the Phones file and not from keyboard.

```
$ cat < Phones
[ contents of Phones ]
$ grep "Nauman" < Phones
[ output of grep ]
$
```

Output Redirection: Here is the syntax for output redirection:

```
command > output-file
```

or

```
command 1> output-file
```

With this command syntax, the display screen is detached from stdout and 'output-file' is attached to it, i.e., 'command' sends output to 'output-file' and not the display screen. Note that 1> operator cannot be used with the C and TC shells. Here is an example use of input redirection. In these examples, the `cat`, `grep`, and `find` commands send their outputs to the Phones, Ali.Phones, and foo.log files, respectively, and not to the display screen.

```
$ cat > Phones
[ your input ]
<Ctrl-D>
$ grep "Ali" Phones > Ali.phones
[ output of grep ]
$ find ~ -name foo -print > foo.log
[ error messages ]
$
```

Error Redirection: Here is the syntax for error redirection:

```
command 2> error-file
```

With this command syntax, the display screen is detached from stderr and 'error-file' is attached to it, i.e., error messages are sent to 'error-file' and not the display screen. Note that 2> cannot be used under C and TC shells. The following are a few examples of error redirection. In these examples, the first `find` command sends its error messages to the errors file and the second `find` command sends its error messages to the /dev/null file. The `ls` command sends its error messages to the error.log file and not to the display screen.

```
$ find ~ -name foo -print 2> errors
[ output of the find command ]
$ ls -l foo 2> error.log
[ output of the find command ]
$ cat error.log
ls: foo: No such file or directory
$ find / -name ls -print 2> /dev/null
/bin/ls
$
```

UNIX/Linux FIFOs

A named pipe (also called a named FIFO, or just FIFO) is a pipe whose access point is a file kept on the file system. By opening this file for reading, a process gets access to the FIFO for reading. By opening the file for writing, the process gets access to the FIFO for writing. By default, a FIFO is opened for blocking I/O. This means that a process reading from a FIFO blocks until another process writes some data in the FIFO. The same goes the other way around. Unnamed pipes can only be used between processes that have an ancestral relationship. And they are temporary; they need to be created every time and are destroyed when the corresponding processes exit. Named pipes (FIFOs) overcome both of these limitations. Figure 10.1 shows two unrelated processes, P1 and P2, communicating with each other using a FIFO.

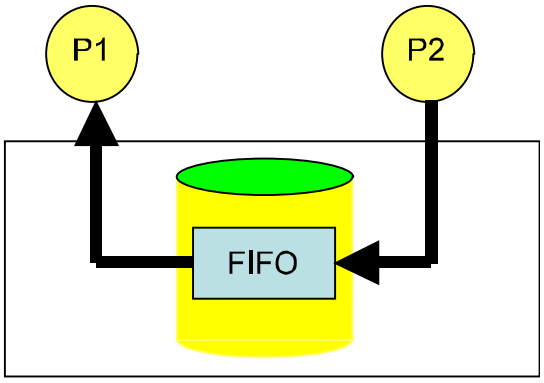

Figure 10.1 Communication between two related or unrelated processes on the same UNIX/Linux machine

Named pipes are created via the `mknod()` system call or `mkfifo()` C library call or by the `mkfifo` command. Here is the synopsis of the mknod() system call.

```
#include <sys/types.h>
#include <sys/stat.h>
int mknod (const char *path, mode_t mode, dev_t dev);
```

The `mknod()` call is normally used for creating special (i.e., device) files but it can be used to create FIFOs too. The 'mode' argument should be permission mode OR-ed with S_IFIFO and 'dev' is set to 0 for creating a FIFO. As is the case with all system calls in UNIX/Linux, `mknod()` returns –1 on failure and *errno* is set accordingly. Some of the reasons for this call to fail are:

- File with the given name exists
- Pathname too long
- A component in the pathname not searchable, non-existent, or non-directory
- Destination directory is read-only
- Not enough memory space available
- Signal caught during the execution of `mknod()`

Here is the synopsis of the `mkfifo()` library call.

```
#include <sys/types.h>
#include <sys/stat.h>
int mkfifo (const char *path, mode_t mode)
```

The argument path is for the name and path of the FIFO created, where was the argument mode is for specifying the file permissions for the FIFO. The specification of the mode argument for this function is the same as for the `open()`. Once we have created a FIFO using `mkfifo()`, we open it using `open()`. In fact, the normal file I/O system calls (`close()`, `read()`, `write()`, `unlink()`, etc.) all works with FIFOs. Since `mkfifo()` invokes the `mknod()` system call, the reasons for its failure are pretty much the same as for the `mknod()` call given above.

Unlike a pipe, a FIFO must be opened before using it for communication. A write to a FIFO that no process has opened for reading results in a SIGPIPE signal. When the last process to write to a FIFO closes it, an EOF is sent to the reader. Multiple processes can write to a FIFO are atomic writes to prevent interleaving of multiple writes.

Two common uses of FIFOs are:

- In client-server applications, FIFOs are used to pass data between a server process and client processes
- Used by shell commands to pass data from one shell pipeline to another, without creating temporary files

In client-server software designed for use on the same machine, the server process creates a "well-known" FIFO. Clients communicate send their requests to the server process via the well-known FIFO. Server sends its response to a client via the client-specific FIFO that each client creates and informs the server process about it. Figure 10.2 shows the diagrammatic view of such a software model.

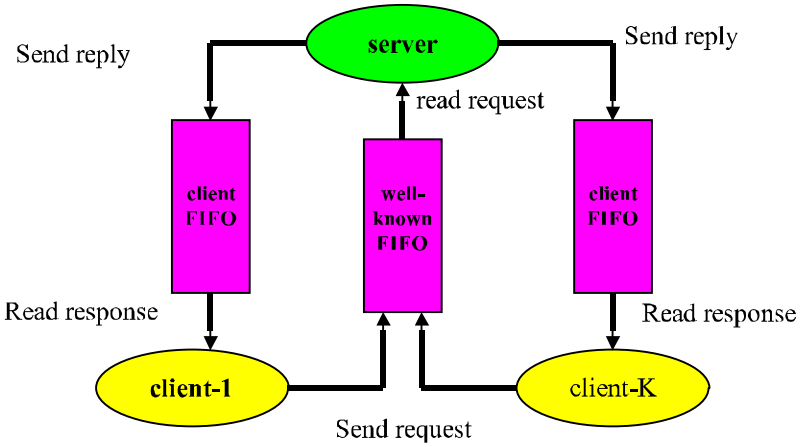

Figure 10.2 Use of FIFOs to implement client-server software on a UNIX/Linux machine

On the command line, `mkfifo` may be used as shown in the following session. As shown in Figure 10.3, the semantics of this session are that prog1 reads its inputs from infile and its output is sent to prog2 and prog3.

```
$ mkfifo fifo1
$ prog3 < fifo1 &
$ prog1 < infile | tee fifo1 | prog2
[ Output ]
$
```

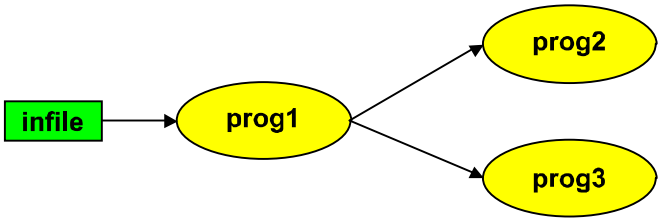

Figure 10.3 Semantics of the above shell session

In the following session, we demonstrate the command line use of FIFOs. The semantics of this session are shown in Figure 10.4. The output of the second command line is the number of lines in the ls.dat file containing ls (i.e., the number of lines in the manual page of the `ls` command containing the string ls) and the output of the third command line is the number of lines in the ls.dat file (i.e., the number of lines in the manual page for the `ls` command).

```
$ man ls > ls.dat
$ cat < fifo1 | grep ls | wc -l &
[1] 21108
$ sort < ls.dat | tee fifo1 | wc -l
  31
  528
$
```

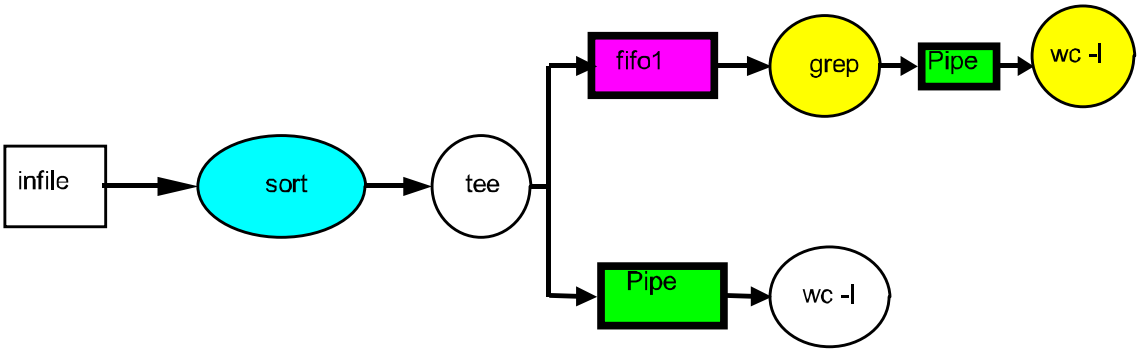

Figure 10.4 Pictorial representation of the semantics of the above shell session

Operating Systems
Lecture No. 11

Reading Material

- UNIX/Linux manual pages for the `mknod()` system call, the `mkfifo()` library call, and the `mkfifo`, `ps`, and `top` commands
- Lecture 11 on Virtual TV

Summary

- More on the use of FIFOs in a program
- Example code for a client-server model
- A few UNIX/Linux process management commands

Use of FIFOs

We continue to discuss the API for using FIFOs for IPC between UNIX/Linux processes. We call these processes client and server. The server process creates two FIFOs, FIFO1 and FIFO2, and opens FIFO1 for reading and FIFO2 for writing. The client process opens FIFO1 for writing and FIFO2 for reading. The client process writes a message to the server process and waits for a response from the server process. The server process reads the message sent by the client process and displays it on the monitor screen. It then sends a message to the client through FIFO2, which the client reads and displays on the monitor screen. The server process then closes the two FIFOs and terminates. The client, after displaying server's message, deletes the two FIFOs and terminates. The protocol for the client-server interaction is shown in Figure 10.1.

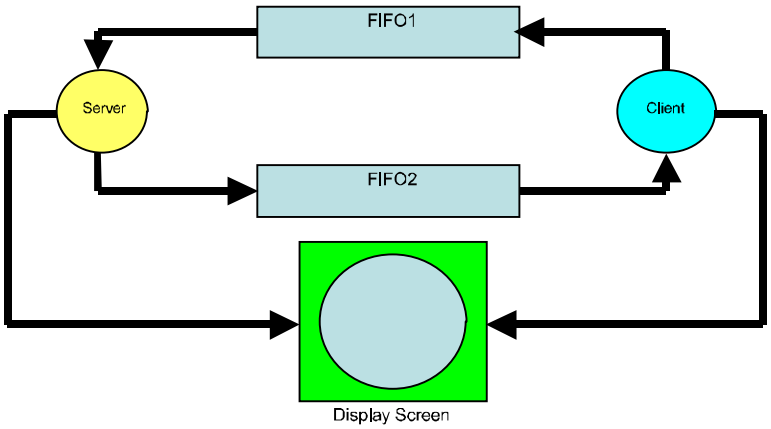

Figure 10.1 Client-server communication using UNIX/Linux FIFOs

The codes for the server and client processes are shown in Figure 10.2 and Figure 10.3, respectively.

```
#include <stdio.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/errno.h>

extern int      errno;

#define FIFO1   "/tmp/fifo.1"
#define FIFO2   "/tmp/fifo.2"
#define PERMS   0666
#define MESSAGE1        "Hello, world!\n"
#define MESSAGE2        "Hello, class!\n"
#include "fifo.h"
main()
{
    char buff[BUFSIZ];
    int readfd, writefd;
    int n, size;

    if ((mknod (FIFO1, S_IFIFO | PERMS, 0) < 0)
                        && (errno != EEXIST)) {
        perror ("mknod FIFO1");
        exit (1);
    }
    if (mkfifo(FIFO2, PERMS) < 0) {
        unlink (FIFO1);
        perror("mknod FIFO2");
        exit (1);
    }
    if ((readfd = open(FIFO1, 0)) < 0) {
        perror ("open FIFO1");
        exit (1);
    }
    if ((writefd = open(FIFO2, 1)) < 0) {
        perror ("open FIFO2");
        exit (1);
    }
    size = strlen(MESSAGE1) + 1;
    if ((n = read(readfd, buff, size)) < 0) {
        perror ("server read"); exit (1);
    }
    if (write (1, buff, n) < n) {
        perror("server write1"); exit (1);
    }
    size = strlen(MESSAGE2) + 1;
    if (write (writefd, MESSAGE2, size) != size) {
        perror ("server write2"); exit (1);
    }
    close (readfd); close (writefd);
}
```

Figure 10.2 Code for the server process

```
#include "fifo.h"
main()
{
    char buff[BUFSIZ];
    int readfd, writefd, n, size;

    if ((writefd = open(FIFO1, 1)) < 0) {
        perror ("client open FIFO1"); exit (1);
    }
    if ((readfd = open(FIFO2, 0)) < 0) {
        perror ("client open FIFO2"); exit (1);
    }
    size = strlen(MESSAGE1) + 1;
    if (write(writefd, MESSAGE1, size) != size) {
        perror ("client write1"); exit (1);
    }
    if ((n = read(readfd, buff, size)) < 0) {
        perror ("client read"); exit (1);
    }
    else
        if (write(1, buff, n) != n) {
            perror ("client write2"); exit (1);
        }
    close(readfd); close(writefd);
     /* Remove FIFOs now that we are done using them */
    if (unlink (FIFO1) < 0) {
        perror("client unlink FIFO1");
        exit (1);
    }
    if (unlink (FIFO2) < 0) {
        perror("client unlink FIFO2");
        exit (1);
    }
    exit (0);
}
```

Figure 10.3 Code for the client process

In the session shown in Figure 10.4, we show how to compile and run the client-server software. We run the server process first so it could create the two FIFOs to be used for communication between the two processes. Note that the server process is run in the background by terminating its command line with an ampersand (&).

```
$ gcc server.c -o server
$ gcc client.c -o client
$ ./server &
[1] 432056
$ ./client
Hello, world!
Hello, class!
$
```

Figure 10.4 Compilation and execution of the client-server software

UNIX/Linux Command for Process Management

We now discuss some of the UNIX/Linux commands for process management, including `ps` and `top`. More commands will be discussed in lecture 12.

`ps` – Display status of processes

`ps` gives a snapshot of the current processes. Without options, ps prints information about processes owned by the user. Some of the commonly used options are `-u`, `-e`, and `-l`.

- `-e` selects all processes
- `-l` formats the output in the long format
- `-u` displays the information in user-oriented format

The shell session in Figure 10.5 shows sample use of the `ps` command. The first command shows the processes running in your current session. The second command shows, page by page, the status of all the processes belonging to root. The last command shows the status of all the processes running on your system.

```
$ ps
  PID TTY          TIME CMD
 1321 pts/0    00:00:00 csh
 1345 pts/0    00:00:00 bash
 1346 pts/0    00:00:00 ps
$ ps -u root | more
  PID TTY          TIME CMD
    1 ?        00:00:04 init
    5 ?        00:00:01 kswapd
  712 ?        00:00:00 inetd
  799 ?        00:00:00 cron
  864 ?        00:00:00 sshd
  934 ?        00:00:00 httpd
 1029 tty1     00:00:00 getty
...
$ ps -e | more
   PID TTY          TIME CMD
     1 ?        00:00:04 init
     2 ?        00:00:00 keventd
     3 ?        00:00:00 ksoftirqd_CPU0
     4 ?        00:00:00 ksoftirqd_CPU1
     5 ?        00:00:01 kswapd
     6 ?        00:00:00 kreclaimd
     7 ?        00:00:00 bdflush
     8 ?        00:00:00 kupdated
...
$
```

Figure 10.5 Use of the `ps` command

top – Display CPU usage of processes

top displays information about the top processes (as many as can fit on the terminal or around 20 by default) on the system and periodically updates this information. Raw CPU percentage is used to rank the processes. A sample run of the top command is shown in Figure 10.6. The output of the command also shows the current time, how long the system has been up and running, number of processes running on the system and their status, number of CPUs in the system and their usage, amount of main memory in the system and its usage, and the size of swap space and its usage. The output also shows a lot of information about each process, including process ID, owner's login name, priority, nice value, and size. Information about processes is updated periodically. See the manual page for the top command for more information by using the man top command.

```
$ top
9:42am  up  5:15,  2 users,  load average: 0.00, 0.00, 0.00
55 processes: 54 sleeping, 1 running, 0 zombie, 0 stopped
CPU0 states:  0.0% user,  0.1% system,  0.0% nice, 99.4% idle
CPU1 states:  0.0% user,  0.2% system,  0.0% nice, 99.3% idle
Mem:   513376K av,  237732K used,  275644K free,      60K shrd,   17944K buff
Swap:  257032K av,       0K used,  257032K free                  106960K cached

  PID USER     PRI  NI  SIZE  RSS SHARE STAT %CPU %MEM   TIME COMMAND
 1406 sarwar    19   0   896  896   700 R     0.3  0.1   0:00 top
 1382 nobody    10   0   832  832   660 S     0.1  0.1   0:00 in.telnetd
    1 root       9   0   536  536   460 S     0.0  0.1   0:04 init
    2 root       9   0     0    0     0 SW    0.0  0.0   0:00 keventd
...
$
```

Figure 10.6 Use of the top command

Operating Systems
Lecture No. 12

Reading Material

- UNIX/Linux manual pages for `fg`, `bg`, `jobs`, and `kill` commands
- Chapter 5 of the textbook
- Lecture 12 on Virtual TV

Summary

- Process Management commands and key presses: `fg`, `bg`, `jobs`, and `kill` commands and `<Ctrl-Z>` and `<Ctrl-C>` command presses
- Thread Concept (thread, states, attributes, etc)

Process Management commands

In the last lecture, we started discussing a few UNIX/Linux process management command. In particular, we discussed the `ps` and `top` commands. We now discuss the `fg`, `bg`, `jobs`, and `kill` commands and `<Ctrl-Z>` and `<Ctrl-C>` key presses.

Moving a process into foreground

You can use the `fg` command to resume the execution of a suspended job in the foreground or move a background job into the foreground. Here is the syntax of the command.

```
fg [%job_id]
```

where, job_id is the job ID (not process ID) of the suspended or background process. If %job_id is omitted, the current job is assumed.

Moving a process into background

You can use the bg command to put the current or a suspended process into the background. Here is the syntax of the command.

```
bg [%job_id]
```

If %job_id is omitted the current job is assumed.

Displaying status of jobs (background and suspended processes)

You can use the jobs command to display the status of suspended and background processes.

Suspending a process

You can suspend a foreground process by pressing `<Ctrl-Z>`, which sends a STOP/SUSPEND signal to the process. The shell displays a message saying that the job has been suspended and displays its prompt. You can then manipulate the state of this

job, put it in the background with the `bg` command, run some other commands, and then eventually bring the job back into the foreground with the `fg` command.

The following session shows the use of the above commands. The `<Ctrl-Z>` command is used to suspend the `find` command and the `bg` command puts it in the background. We then use the `jobs` command to display the status of jobs (i.e., the background or suspended processes). In our case, the only job is the find command that we explicitly put in the background with the `<Ctrl-Z>` and `bg` commands.

```
$ find / -name foo -print 2> /dev/null
^Z
[1]+  Stopped    find / -name foo -print 2> /dev/null
$ bg
[1]+ find / -name foo -print 2> /dev/null &
$ jobs
[1]+  Running    find / -name foo -print 2> /dev/null &
$ fg
find / -name foo -print 2> /dev/null
[ command output ]
$
```

Terminating a process

You can terminate a foreground process by pressing `<Ctrl-C>`. Recall that this key press sends the SIGINT signal to the process and the default action is termination of the process. Of course, if your foreground process intercepts SIGINT and ignores it, you cannot terminate it with `<Ctrl-C>`. In the following session, we terminate the find command with `<Ctrl-C>`.

```
$ find / -name foo -print 1> out 2> /dev/null
^C
$
```

You can also terminate a process with the kill command. When executed, this command sends a signal to the process whose process ID is specified in the command line. Here is the syntax of the command.

```
kill [-signal] PID
```

where, 'signal' is the signal number and PID is the process ID of the process to whom the specified signal is to be sent. For example, `kill -2 1234` command sends signal number 2 (which is also called SIGINT) to the process with ID 1234. The default action for a signal is termination of the process identified in the command line. When executed without a signal number, the command sends the SIGTERM signal to the process. A process that has been coded to intercept and ignore a signal, can be terminated by sending it the 'sure kill' signal, SIGKILL, whose signal number is 9, as in `kill -9 1234`.

You can display all of the signals supported by your system, along with their numbers, by using the `kill -l` command. On some systems, the signal numbers are not displayed. Here is a sample run of the command on Solaris 2.

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGEMT       8) SIGFPE
 9) SIGKILL     10) SIGBUS      11) SIGSEGV     12) SIGSYS
13) SIGPIPE     14) SIGALRM     15) SIGTERM     16) SIGUSR1
...
$
```

The Thread Concept

There are two main issues with processes:

1. The `fork()` system call is expensive (it requires memory to memory copy of the executable image of the calling process and allocation of kernel resources to the child process)
2. An inter-process communication channel (IPC) is required to pass information between a parent process and its children processes.

These problems can be overcome by using threads.

A thread, sometimes called a **lightweight process (LWP)**, is a basic unit of CPU utilization and executes within the address space of the process that creates it. It comprises a thread ID, a program counter, a register set, errno, and a stack. It shares with other threads belonging to the same process its code sections, data section, current working directory, user and group IDs, signal setup and handlers, PCB and other operating system resources, such as open files and system. A traditional (heavy weight) process has a single thread of control. If a process has multiple threads of control, it can do more than one task at a time. Figure 12.1 shows processes with single and multiple threads. Note that, as stated above, threads within a process share code, data, and open files, and have their own register sets and stacks.

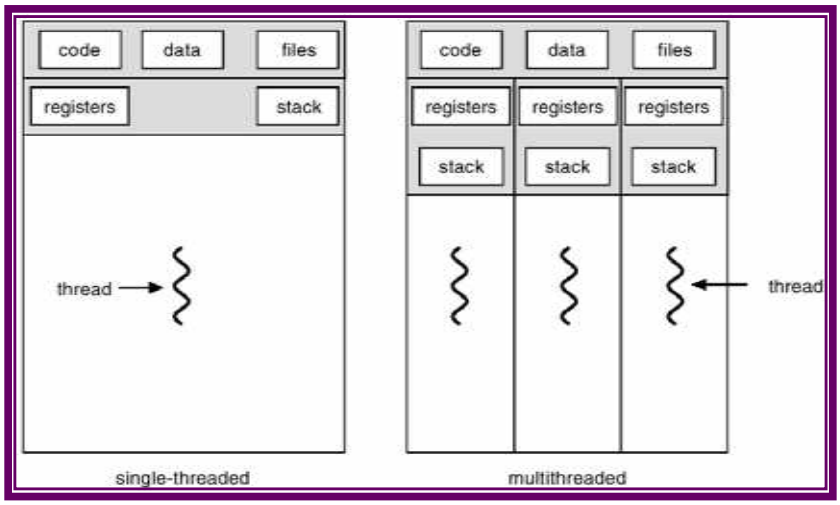

Figure 12.1 Single- and multi-threaded processes

In Figure 12.2, we show the code structure for a sequential (single-threaded) process and how the control thread moves from the main function to the f1 function and back, and from f1 to main and back. The important point to note here is that there is just one thread of control that moves around between various functions.

```
main()
{

     f1(

     f2(...);
     ...
}
f1(...)
{  ...  }
f2(...)
{  ...  }
```

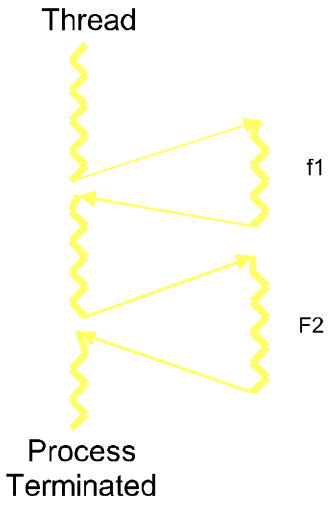

Figure 12.2 Code structure of a single-threaded (sequential) process

In Figure 12.3, we show the code structure for a multi-threaded process and how multiple threads of control are active simultaneously. We use hypothetical function `thread()` to create a thread. This function takes two arguments: the name of a function for which a thread has to be created and a variable in which the ID of the newly created thread is to be stored. The important point to note here is that multiple threads of control are simultaneously active within the same process; each thread steered by its own PC.

```
main()
{
     ...
     thread(t1,f1);
     ...
     thread(t2,f2);
     ...
}
f1(...)
{  ...  }
f2(...)
{  ...  }
```

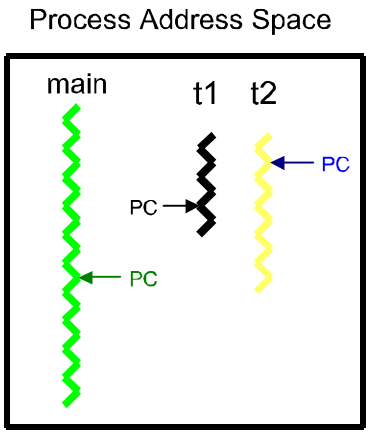

Figure 12.3 Code structure of a multi-threaded process

The Advantages and Disadvantages of Threads

Four main advantages of threads are:

1. Responsiveness: Multithreading an interactive application may allow a program to continue running even if part of it is blocked or is performing a lengthy operation, thereby increasing responsiveness to the user.
2. Resource sharing: By default, threads share the memory and the resources of the process to which they belong. Code sharing allows an application to have several different threads of activity all within the same address space.
3. Economy: Allocating memory and resources for process creation is costly. Alternatively, because threads share resources of the process to which they belong, it is more economical to create and context switch threads.
4. Utilization of multiprocessor architectures: The benefits of multithreading of multithreading can be greatly increased in a multiprocessor environment, where each thread may be running in parallel on a different processor. A single threaded process can only run on one CPU no matter how many are available. Multithreading on multi-CPU machines increases concurrency.

Some of the main disadvantages of threads are:

1. Resource sharing: Whereas resource sharing is one of the major advantages of threads, it is also a disadvantage because proper synchronization is needed between threads for accessing the shared resources (e.g., data and files).
2. Difficult programming model: It is difficult to write, debug, and maintain multi-threaded programs for an average user. This is particularly true when it comes to writing code for synchronized access to shared resources.

Operating Systems
Lecture No. 13

Reading Material

- UNIX/Linux manual pages `pthread_create()`, `pthread_join()`, and `pthread_exit()` calls
- Chapter 5 of the textbook
- Lecture 13 on Virtual TV

Summary

- User- and Kernel –level threads
- Multi-threading models
- Solaris 2 threads model
- POSIX threads (the pthread library)
- Sample code

User and Kernel Threads

Support for threads may be provided at either user level for *user threads* or by kernel for *kernel threads.*

User threads are supported above kernel and are implemented by a thread library at the user level. The library provides support for thread creation, scheduling, and management with no support from the kernel. Since the kernel is unaware of user-level threads, all thread creation and scheduling are done in the user space without the need for kernel intervention, and therefore are fast to create and manage. If the kernel is single threaded, then any user level thread performing a blocking system call will cause the entire process to block, even if other threads are available to run within the application. User thread libraries include POSIX Pthreads , Solaris 2 UI-threads, and Mach C-threads.

Kernel threads are supported directly by the operating system. The kernel performs the scheduling, creation, and management in kernel space; the kernel level threads are hence slower to create and manage, compared to user level threads. However since the kernel is managing threads, if a thread performs a blocking system call, the kernel can schedule another thread in the application for execution. Windows NT, Windowss 2000, Solaris, BeOS and Tru64 UNIX support kernel threads.

Multi-threading Models

There are various models for mapping user-level threads to kernel-level threads. We describe briefly these models, their main characteristics, and examples.

1. **Many-to-One**: In this model, many user-level threads are supported per kernel thread, as shown in Figure 13.1. Since only one kernel-level thread supports many user threads, there is no concurrency. This means that a process blocks when a thread makes a system call. Examples of these threads are Solaris Green threads POSIX Pthreads.

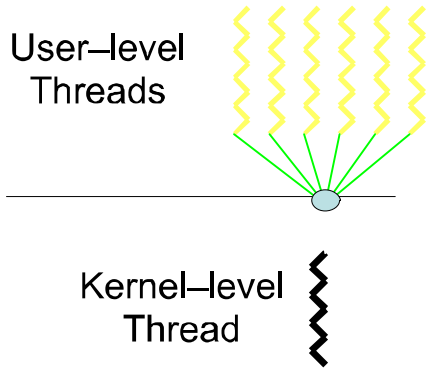

Figure 13.1 Many –to-One Model

2. **One-to-One**: In this model, there is a kernel thread for every user thread, as shown in Figure 13.2. Thus, this model provides true concurrency. This means that a process does not block when a thread makes a system call. The main disadvantage of this model is the overhead of creating a kernel thread per user thread. Examples of these threads are WindowsNT, Windows 2000, and OS/2.

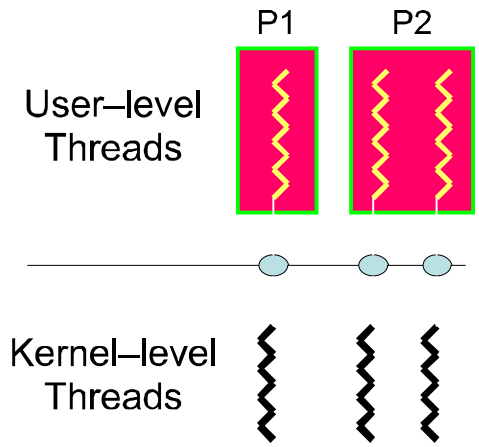

Figure 13.2 One-to-One Model

3. **Many-to-One**: In this model, multiple user-level threads are multiplexed over a smaller or equal number of kernel threads, as shown in Figure 13.2. Thus, true concurrency is not achieved through this model. Examples of these threads are Solais 2 and HP-UX.

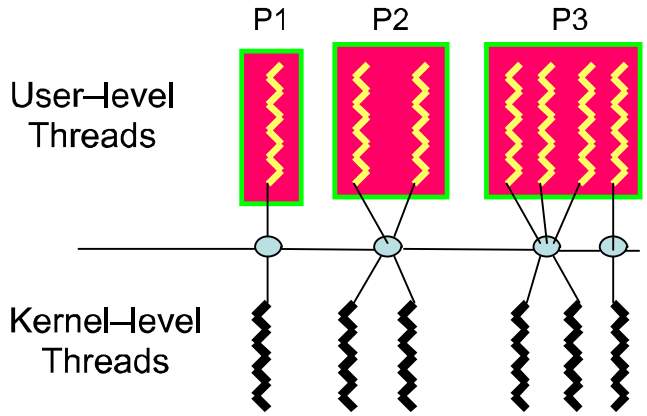

Figure 13.3 Many-to Many Model

Solaris 2 Threads Model

Solaris 2 has threads, lightweight processes (LWPs), and processes, as shown in Figure 13.4. At least one LWP is assigned to every user process to allow a user thread to talk to a kernel thread. User level threads are switched and scheduled among LWPs without kernel's knowledge. One kernel thread is assigned per LWP. Some kernel threads have no LWP associated with them because these threads are not executed for servicing a request by a user-level thread. Examples of such kernel threads are clock interrupt handler, swapper, and short-term (CPU) scheduler.

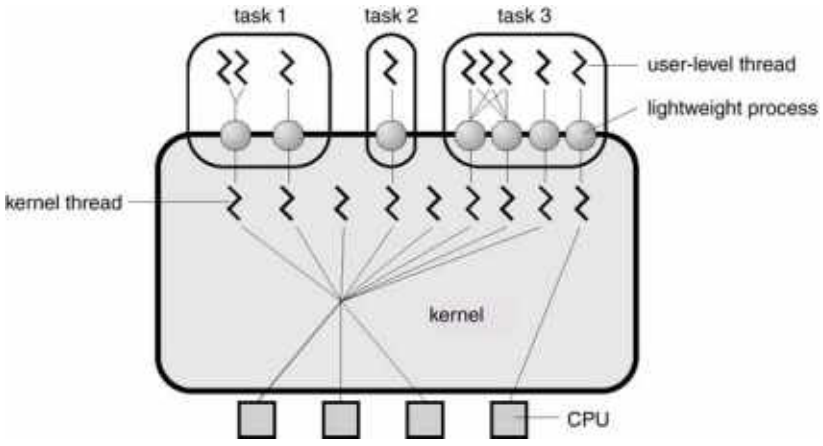

Figure 13.4 Solaris 2 Threads Model

POSIX Threads (the pthread library)

Pthreads refers to the POSIX standard defining an API for thread creation, scheduling, and synchronization. This is a specification for thread behavior not an implementation. OS designers may implement the specification in any way they wish. Generally, libraries implementing the Pthreads specification are restricted to UNIX-based systems such as Solaris 2. In this section, we discuss the Pthreads library calls for creating, joining, and terminating threads and use these calls to write small multi-threaded C programs.

Creating a Thread

You can create a threads by using the `pthread_create()` call. Here is the syntax of this call.

```
int pthread_create(pthread_t *threadp, const pthread_attr_t *attr,
                   void* (*routine)(void *), arg *arg);
```

where, 'threadp' contains thread ID (TID) of the thread created by the call, 'attr' is used to modify the thread attributes (stack size, stack address, detached, joinable, priority, etc.), 'routine' is the thread function, and 'arg' is any argument we want to pass to the thread function. The argument does not have to be a simple native type; it can be a 'struct' of whatever we want to pass in.

The `pthread_create()` call fails and returns the corresponding value if any of the following conditions is detected:

- **EAGAIN** The system-imposed limit on the total number of threads in a process has been exceeded or some system resource has been exceeded (for example, too many LWPs were created).
- **EINVAL** The value specified by 'attr' is invalid.
- **ENOMEM** Not enough memory was available to create the new thread.

You can do error handling by including the <errno.h> file and incorporating proper error handling code in your programs.

Joining a Thread

You can have a thread wait for another thread within the same process by using the pthread_join() call. Here is the syntax of this call.

```
int pthread_join(pthread_t aThread, void **statusp);
```

where, 'aThread' is the thread ID of the thread to wait for and 'statusp' gets the return value of `pthread_exit()` call made in the process for whom wait is being done.

A thread can only wait for a joinable thread in the same process address space; a thread cannot wait for a detached thread. Multiple threads can join with a thread but only one returns successfully; others return with an error that no thread could be found with the given TID

Terminating a Thread

You can terminate a thread explicitly by either returning from the thread function or by using the `pthread_exit()` call. Here is the syntax of the `pthread_exit()` call.

```
void pthread_exit(void *valuep);
```

where, 'valuep' is a pointer to the value to be returned to the thread which is waiting for this thread to terminate (i.e., the thread which has executed `pthread_join()` for this thread).

A thread also terminates when the main thread in the process terminates. When a thread terminates with the exit() system call, it terminates the whole process because the purpose of the exit() system call is to terminate a process and not a thread.

Sample Code

The following code shows the use of the pthread library calls discussed above. The program creates a thread and waits for it. The child thread displays the following message on the screen and terminates.

```
Hello, world! ... The threaded version.
```

As soon as the child thread terminates, the parent comes out of wait, displays the following message and terminates.

```
Exiting the main function.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
/* Prototype for a function to be passed to our thread */
void* MyThreadFunc(void *arg);
int main()
{
   pthread_t aThread;
   /* Create a thread and have it run the MyThreadFunction */
   pthread_create(&aThread, NULL, MyThreadFunc, NULL);
   /* Parent waits for the aThread thread to exit */
   pthread_join(aThread, NULL);
   printf ("Exiting the main function.\n");
   return 0;
}
void* MyThreadFunc(void* arg)
{
        printf ("Hello, world! ... The threaded version.\n");
        return NULL;
}
```

The following session shows compilation and execution of the above program. Does the output make sense to you?

```
$ gcc hello.c -o hello -lpthread -D_REENTRANT
$ hello
Hello, world! ... The threaded version.
Exiting the main function.
$
```

Note that you need to take the following steps in order to be able to use the pthread library.

1. Include <pthread.h> in your program
2. Link the pthread library with your program (by using the –lpthread option in the compiler command)
3. Pass the _REENTRANT macro from the command line (or define it in your program)

Here is another program that uses the pthread library to create multiple threads and have them display certain messages. Read through the code to understand what it does. Then compile and run it on your UNIX/Linux system to display output of the program and to see if you really understood the code.

```
/******************************************************************************
* FILE: hello_arg2.c
* DESCRIPTION:
*   A "hello world" Pthreads program which demonstrates another safe way
*   to pass arguments to threads during thread creation.  In this case,
*   a structure is used to pass multiple arguments.
*
* LAST REVISED: 09/04/02 Blaise Barney
******************************************************************************/
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS   8

char *messages[NUM_THREADS];

struct thrcad_data
{
  int       thread_id;
  int  sum;
  char *message;
};

struct thread_data thread_data_array[NUM_THREADS];

void *PrintHello(void *threadarg)
{
  int taskid, sum;
  char *hello_msg;
  struct thread_data *my_data;

  sleep(1);
  my_data = (struct thread_data *) threadarg;
  taskid = my_data->thread_id;
  sum = my_data->sum;
  hello_msg = my_data->message;
  printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
  pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
```

```
  pthread_t threads[NUM_THREADS];
  int *taskids[NUM_THREADS];
  int rc, t, sum;

  sum=0;
  messages[0] = "English: Hello World!";
  messages[1] = "French: Bonjour, le monde!";
  messages[2] = "Spanish: Hola al mundo";
  messages[3] = "Klingon: Nuq neH!";
  messages[4] = "German: Guten Tag, Welt!";
  messages[5] = "Russian: Zdravstvytye, mir!";
  messages[6] = "Japan: Sekai e konnichiwa!";
  messages[7] = "Latin: Orbis, te saluto!";

  for(t=0; t<NUM_THREADS; t++) {
    sum = sum + t;
    thread_data_array[t].thread_id = t;
    thread_data_array[t].sum = sum;
    thread_data_array[t].message = messages[t];
    printf("Creating thread %d\n", t);
    rc = pthread_create(&threads[t], NULL, PrintHello, (void *) &thread_data_array[t]);
    if (rc) {
      printf("ERROR; return code from pthread_create() is %d\n", rc);
      exit(-1);
    }
  }
  pthread_exit(NULL);
}
```

Reference

The above code was taken from the following website.

`http://www.llnl.gov/computing/tutorials/pthreads/samples/hello_arg2.c`

Operating Systems
Lecture No. 14

Reading Material

- Chapter 6 of the textbook
- Lecture 14 on Virtual TV

Summary

- Basic concepts
- Scheduling criteria
- Preemptive and non-preemptive algorithms
- First-Come-First-Serve scheduling algorithm

Basic Concepts

The objective of multiprogramming is to have some process running at all times, in order to maximize CPU utilization. In a uniprocessor system, only one process may run at a time; any other processes much wait until the CPU is free and can be rescheduled.
In multiprogramming, a process is executed until it must wait, typically for the completion of some I/O request. In a simple computer system, the CPU would then sit idle; all this waiting time is wasted. Multiprogramming entails productive usage of this time. When one process has to wait, the OS takes the CPU away from that process and gives the CPU to another process. Almost all computer resources are scheduled before use.

Life of a Process

As shown in Figure 14.1, process execution consists of a cycle of CPU execution and I/O wait. Processes alternates between these two states. Process execution begins with a **CPU burst.** An I/O burst follows that, and so on**.** Eventually, the last CPU burst will end with a system request to terminate execution, rather than with another I/O burst.

An I/O bound program would typically have many very short CPU bursts. A CPU-bound program might have a few very long CPU bursts. This distribution can help us select an appropriate CPU-scheduling algorithm. Figure 14.2 shows results on an empirical study regarding the CPU bursts of processes. The study shows that most of the processes have short CPU bursts of 2-3 milliseconds.

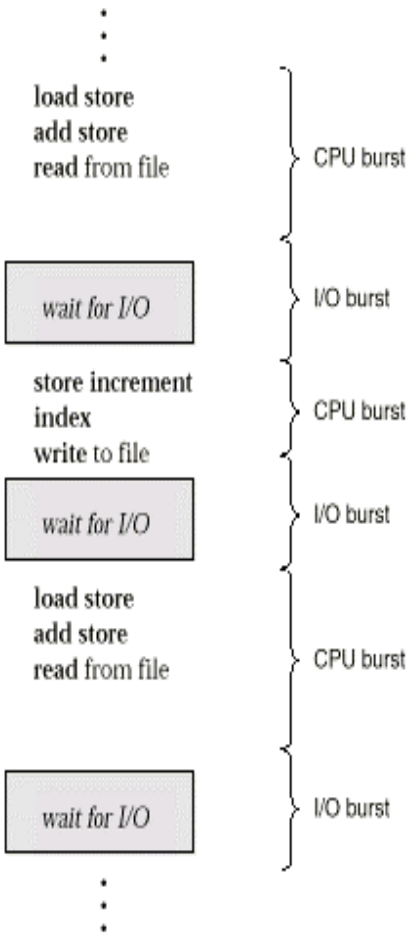

Figure 14.1 Alternating Sequence of CPU and I/O Bursts

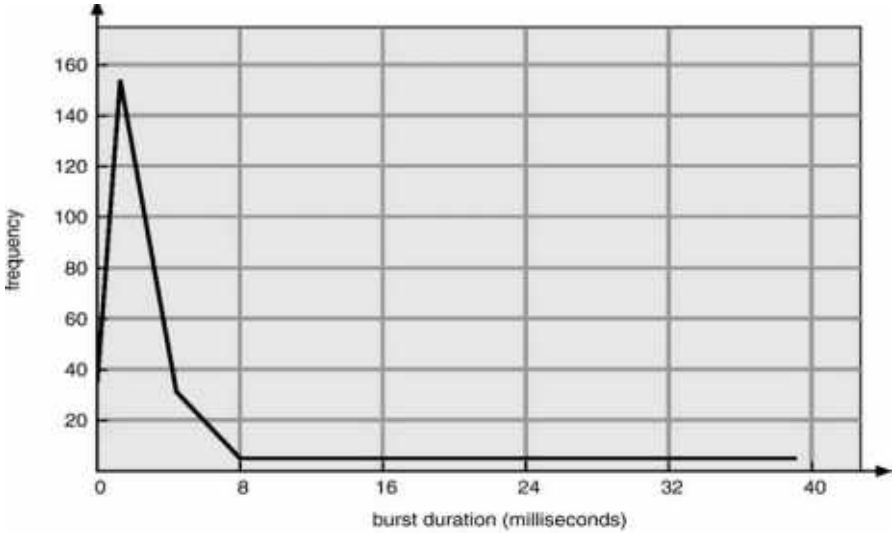

Figure 14.2 Histogram of CPU-burst Times

CPU Scheduler

Whenever the CPU becomes idle, the operating system must select one of the processes in the ready queue to be executed. The **short-term scheduler** (i.e., the CPU scheduler) selects a process to give it the CPU. It selects from among the processes in memory that are ready to execute, and invokes the dispatcher to have the CPU allocated to the selected process.

A ready queue can be implemented as a FIFO queue, a tree, or simply an unordered linked list. The records (nodes) in the ready queue are generally the process control blocks (PCBs) of processes.

Dispatcher

The **dispatcher** is a kernel module that takes control of the CPU from the current process and gives it to the process selected by the short-term scheduler. This function involves:

- Switching the context (i.e., saving the context of the current process and restoring the context of the newly selected process)
- Switching to user mode
- Jumping to the proper location in the user program to restart that program

The time it takes for the dispatcher to stop one process and start another running is known as the **dispatch latency**.

Preemptive and Non-Preemptive Scheduling

CPU scheduling can take place under the following circumstances:

1. When a process switches from the running state to the waiting state (for example, an I/O request is being completed)
2. When a process switches from the running state to the ready state (for example when an interrupt occurs)
3. When a process switches from the waiting state to the ready state (for example, completion of I/O)
4. When a process terminates

In 1 and 4, there is no choice in terms of scheduling; a new process must be selected for execution. There is a choice in case of 2 and 3. When scheduling takes place only under 1 and 4, we say, scheduling is **non-preemptive**; otherwise the scheduling scheme is **preemptive**. Under non-preemptive scheduling once the CPU has been allocated to a process the process keeps the CPU until either it switches to the waiting state, finishes its CPU burst, or terminates. This scheduling method does not require any special hardware needed for preemptive scheduling.

Preemptive scheduling incurs a cost. Consider the case of two processes sharing data. One may be in the midst of updating the data when it is preempted and the second process is run. The second process may try to read the data, which are currently in an inconsistent state. New mechanisms are needed to coordinate access to shared data. We discuss this topic in Chapter 7 of the textbook.

Scheduling Criteria

The scheduling criteria include:

- **CPU utilization**: We want to keep CPU as busy as possible. In a real system it should range from 40 percent (for a lightly loaded system) to 90 percent (for a heavily used system)
- **Throughput:** If CPU is busy executing processes then work is being done. One measure of work is the number of processes completed per time, called, **throughput.** We want to maximize the throughput.
- **Turnaround time:** The interval from the time of submission to the time of completion is the **turnaround time.** Turnaround time is the sum of the periods spent waiting to get into memory, waiting in the ready queue, executing on the CPU and doing I/O. We want to minimize the turnaround time.
- **Waiting time:** Waiting time is the time spent waiting in the ready queue. We want to minimize the waiting time to increase CPU efficiency.
- **Response time:** It is the time from the submission of a request until the first response is produced. Thus response time is the amount of time it takes to start responding but not the time it takes to output that response. Response time should be minimized.

Scheduling Algorithms

We will now discuss some of the commonly used short-term scheduling algorithms. Some of these algorithms are suited well for batch systems and others for time-sharing systems. Here are the algorithms we will discuss:

- First-Come-First-Served (FCFS) Scheduling
- Shorted Job First (SJF) Scheduling
- Shortest Remaining Time First (SRTF) Scheduling
- Priority Scheduling
- Round-Robin Scheduling
- Multilevel Queues Scheduling
- Multilevel Feedback Queues Scheduling
- UNIX System V Scheduling

First-Come, First-Served (FCFS) Scheduling

The process that requests the CPU first (i.e., enters the ready queue first) is allocated the CPU first. The implementation of an FCFS policy is managed with a FIFO queue. When a process enters the ready queue, its PCB is linked onto the tail of the queue. When CPU is free, it is allocated to the process at the head of the queue. The running process is removed from the queue. The average waiting time under FCFS policy is not minimal and may vary substantially if the process CPU-burst times vary greatly. FCFS is a non-preemptive scheduling algorithm.

We use the following system state to demonstrate the working of this algorithm. For simplicity, we assume that processes are in the ready queue at time 0.

Process	Burst Time
P1	24
P2	3
P3	3

Suppose that processes arrive into the system in the order: P1, P2, P3. Processes are served in the order: P1, P2, P3.The **Gantt chart** for the schedule is shown in Figure 14.3.