- How will I/O be handled? Will each thread perform its own I/O for example?

Each of these considerations is important, and to some extent each arises in most programming problems. Determining data dependencies, deciding which data should be shared and which should be private, and determining how to synchronize access to shared data are very critical aspects to the correctness of a solution. Load balancing and the handling of I/O usually affect performance but not correctness.

Knowing how to use a thread library is just the technical part of using threads. The much harder part is knowing how to write a parallel program. These notes are not intended to assist you in that task. Their purpose is just to provide the technical background, with pointers here and there. However, before continuing, we present a few common paradigms for organizing multi-threaded programs.

Thread Pool, or Boss/Worker Paradigm

In this approach, there is a single *boss* thread that dispatches threads to perform work. These threads are part of a worker thread pool which is usually pre-allocated before the boss begins dispatching threads.

Peer or WorkCrew Paradigm

In the WorkCrew model, tasks are assigned to a finite set of worker threads. Each worker can enqueue subtasks for concurrent evaluation by other workers as they become idle. The Peer model is similar to the boss/worker model except that once the worker pool has been created, the boss becomes the another thread in the thread pool, and is thus, a peer to the other threads.

Pipeline

Similar to how pipelining works in a processor, each thread is part of a long chain in a processing factory. Each thread works on data processed by the previous thread and hands it off to the next thread. You must be careful to equally distribute work and take extra steps to ensure non-blocking behavior in this thread model or you could experience pipeline "stalls."

10.4 Overview of the Pthread Library

In 1995 the Open Group defined a standard interface for UNIX threads (IEEE POSIX 1003.1c) which they named *Pthreads* (P for POSIX). This standard was supported on multiple platforms, including Solaris, Mac OS, FreeBSD, OpenBSD, and Linux. In 2005, a new implementation of the interface was developed by Ulrich Drepper and Ingo Molnar of Red Hat, Inc. called the *Native POSIX Thread Library* (*NPTL*), which was much faster than the original library, and has since replaced that library. The Open Group further revised the standard in 2008. We will limit our study of threads to the NPTL implementation of Pthreads.

The Pthreads library provides a very large number of primitives for the management and use of threads; there are 93 different functions defined in the 2008 POSIX standard. Some thread functions are analogous to those of processes. The following table compares the basic process primitives to analogous Pthread primitives.

Process Primitive	**Thread Primitive**	**Description**
`fork()`	`pthread_create()`	Create a new flow of control with a function to execute
`exit()`	`pthread_exit()`	Exit from the calling flow of control
`waitpid()`	`pthread_join()`	Wait for a specific flow of control to exit and collect its status
`getpid()`	`pthread_self()`	Get the id of the calling flow of control
`abort()`	`pthread_cancel()`	Request abnormal termination of the calling flow of control

The Pthreads API can be categorized roughly by the following four groups

Thread management: This group contains functions that work directly on threads, such as creating, detaching, joining, and so on. This group also contains functions to set and query thread attributes.

Mutexes: This group contains functions for handling critical sections using mutual exclusion. Mutex functions provide for creating, destroying, locking and unlocking mutexes. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes.

Condition variables: This group contains functions that address communications between threads that share a mutex based upon programmer-specified conditions. These include functions to create, destroy, wait and signal based upon specified variable values, as well as functions to set and query condition variable attributes.

Synchronization: This group contains functions that manage read/write locks and barriers.

We will visit these groups in the order they are listed here, not covering any in great depth, but enough depth to write fairly robust programs.

10.5 Thread Management

10.5.1 Creating Threads

We will start with the `pthread_create()` function. The prototype is

```
int pthread_create  ( pthread_t            *thread,
                      const pthread_attr_t *attr,
                      void  *(*start_routine)(void *),
                      void                 *arg);
```

This function starts a new thread with thread ID `*thread` as part of the calling process. On successful creation of the new thread, `thread` contains its thread ID. Unlike `fork()`, this call passes the address of a function, `start_routine()`, to be executed by the new thread. This "start" function has exactly one argument, of type `void*`, and returns a `void*`. The fourth argument, `arg`, is the argument that will be passed to `start_routine()` in the thread.

The second argument is a pointer to a `pthread_attr_t` structure. This structure can be used to define attributes of the new thread. These attributes include properties such as its stack size, scheduling policy, and *joinability* (to be discussed below). If the program does not specifically set values for its members, default values are used instead. We will examine thread properties in more detail later.

Because `start_routine()` has just a single argument, if the function needs access to more than a simple variable, the program should declare a structure with all state that needs to be accessed within the thread, and pass a pointer to that structure. For example, if a set of threads is accessing a shared array and each thread will process a contiguous portion of that array, you might want to define a structure such as

```
typedef struct _task_data
{
    int first;    /* index of first element for task */
    int last;     /* index of last element for task */
    int *array;   /* pointer to start of array */
    int task_id;  /* id of thread */
} task_data;
```

and start each thread with the values of `first`, `last`, and `task_id` initialized. The array pointer may or may not be needed; if the array is a global variable, the threads will have access to it. If it is declared in the main program, then its address can be part of the structure. Suppose that the array is declared as a static local variable named `data_array` in the main program. Then a code fragment to initialize the thread data and create the threads could be

```
task_data  thread_data[NUM_THREADS];
for ( t = 0 ; t < NUM_THREADS; t++) {
    thread_data[t].first     = t*size;
    thread_data[t].last      = (t+1)*size -1;
    if ( thread_data[t].last > ARRAY_SIZE -1 )
        thread_data[t].last  = ARRAY_SIZE - 1;
    thread_data[t].array     = &data_array[0];
    thread_data[t].task_id   = t;

    if ( 0 != (rc = pthread_create(&threads[t], NULL, process_array,
                             (void *) &thread_data[t]) ) )  {
        printf("ERROR; %d return code from pthread_create()\n", rc);
        exit(-1);
    }
}
```

This would create `NUM_THREADS` many threads, each executing `process_array`(), each with its own structure containing parameters of its execution.

10.5.1.1 Design Decision Regarding Shared Data

The advantage of declaring the data array as a local variable in the main program is that it makes it easier to analyze and maintain the code when there are fewer global variables and side effects. Programs with functions that modify global variables are harder to analyze. On the other hand, making it a local in main and then having to add a pointer to that array in the thread data structure passed to each thread increases thread storage requirements and slows down the program. Each thread has an extra pointer in its stack when it executes, and each reference to the array requires two dereferences instead of one. Which is preferable? It depends what the overall project requirements are. If speed and memory are a concern, use a global and use good practices in documenting and accessing it. If not, use the static local.

10.5.2 Thread Identification

A thread can get its thread ID by calling `pthread_self()`, whose prototype is

```
pthread_t pthread_self(void);
```

This is the analog to `getpid()` for processes. This function is the only way that the thread can get its ID, because it is not provided to it by the creation call. It is entirely analogous to `fork()` in this respect.

A thread can check whether two thread IDs are equal by calling

```
int pthread_equal(pthread_t t1, pthread_t t2);
```

This returns a non-zero if the two thread IDs are equal and zero if they are not.

10.5.3 Thread Termination

A thread can terminate itself by calling `pthread_exit()`:

```
void pthread_exit(void *retval);
```

This function kills the thread. The `pthread_exit()` function never returns. Analogous to the way that `exit()` returns a value to `wait()`, the return value may be examined from another thread in the same process if it calls `pthread_join()`[1]. The value pointed to by `retval` should not be located on the calling thread's stack, since the contents of that stack are undefined after the thread terminates. It can be a global variable or allocated on the heap. Therefore, if you want to use a locally-scoped variable for the return value, declare it as static within the thread.

It is a good idea for the main program to terminate itself by calling `pthread_exit()`, because if it has not waited for spawned threads and they are still running, if it calls `exit()`, they will be killed. If these threads should not be terminated, then calling `pthread_exit()` from `main()` will ensure that they continue to execute.

[1]Provided that the terminating thread is joinable.

10.5.4 Thread Joining and Joinability

When a thread is created, one of the attributes defined for it is whether it is *joinable* or *detached*. By default, created threads are joinable. If a thread is joinable, another thread can wait for its termination using the function `pthread_join()`. Only threads that are created as joinable can be joined.

Joining is a way for one thread to wait for another thread to terminate, in much the same way that the `wait()` system calls lets a process wait for a child process. When a parent process creates a thread, it may need to know when that thread has terminated before it can perform some task. Joining a thread, like waiting for a process, is a way to synchronize the performance of tasks.

However, joining is different from waiting in one respect: the thread that calls `pthread_join()` must specify the thread ID of the thread for which it waits, making it more like `waitpid()`. The prototype is

```
int pthread_join(pthread_t thread, void **value_ptr);
```

The `pthread_join()` function suspends execution of the calling thread until the target thread terminates, unless the target thread has already terminated. If the target thread already terminated, `pthread_join()` returns successfully.

If `value_ptr` is not `NULL`, then the value passed to `pthread_exit()` by the terminating thread will be available in the location referenced by `value_ptr`, provided `pthread_join()` succeeds.

Some things that cause problems include:

- Multiple simultaneous calls to `pthread_join()` specifying the same target thread have undefined results.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` does not refer to a joinable thread.
- The behavior is undefined if the value specified by the thread argument to `pthread_join()` refers to the calling thread.
- Failing to join with a thread that is joinable produces a "zombie thread". Each zombie thread consumes some system resources, and when enough zombie threads have accumulated, it will no longer be possible to create new threads (or processes).

The following listing shows a simple example that creates a single thread and waits for it using `pthread_join()`, collecting and printing its exit status.

Listing 10.1: Simple example of thread creation with join

```
int exitval;

void* hello_world( void * world)
{
    printf("Hello World from %s.\n", (char*) world);
    exitval = 2;
    pthread_exit((void*) exitval) ;
}
```

```
int main( int argc, char *argv[])
{
    pthread_t  child_thread;
    void  *status;
    char  *planet  = "Pluto";

    if ( 0 != pthread_create(&child_thread, NULL,
                            hello_world,( void*) planet) ) {
        perror("pthread_create");
        exit(-1);
    }
    pthread_join(child_thread, (void**) (&status));
    printf("Child exited with status %ld\n", (long) status);
    return 0;
}
```

Any thread in a process can join with any other thread. They are peers in this sense. The only obstacle is that to join a thread, it needs its thread ID.

10.5.5 Detached Threads

Because `pthread_join()` must be able to retrieve the status and thread ID of a terminated thread, this information must be stored someplace. In many Pthread implementations, it is stored in a structure that we will call a *Thread Control Block* (TCB). In these implementations, the entire TCB is kept around after the thread terminates, just because it is easier to do this. Therefore, until a thread has been joined, this TCB exists and uses memory. Failing to join a joinable thread turns these TCBs into waste memory.

Sometimes threads are created that do not need to be joined. Consider a process that spawns a thread for the sole purpose of writing output to a file. The process does not need to wait for this thread. When a thread is created that does not need to be joined, it can be created as a *detached thread*. When a detached thread terminates, no resources are saved; the system cleans up all resources related to the thread.

A thread can be created in a detached state, or it can be detached after it already exists. To create a thread in a detached state, you can use the `pthread_attr_setdetachstate()` function to modify the `pthread_attr_t` structure prior to creating the thread, as in:

```
pthread_t      tid;  /* thread ID        */
pthread_attr_t attr; /* thread attribute */

pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

/* now create the thread */
pthread_create(&tid, &attr, start_routine, arg);
```

An existing thread can be detached using `pthread_detach()`:

```
int pthread_detach(pthread_t thread);
```

The function `pthread_detach()` can be called from any thread, in particular from within the thread itself! It would need to get its thread ID using `pthread_self()`, as in

```
pthread_detach(pthread_self());
```

Once a thread is detached, it cannot become joinable. It is an irreversible decision. The following listing shows how a main program can exit, using `pthread_exit()` to allow its detached child to run and produce output, even after `main()` has ended. The call to `usleep()` gives a bit of a delay to simulate computationally demanding output being produced by the child.

Listing 10.2: Example of detached child

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

void *thread_routine(void * arg)
{
    int   i;
    int   bufsize = strlen(arg);
    int   fd = 1;

    printf("Child is running...\n");
    for (i = 0; i < bufsize; i++) {
             usleep(500000);
             write(fd, arg+i, 1);
    }
    printf("\nChild is now exiting.\n");
    return(NULL);
}

int main(int argc, char* argv[])
{
    char * buf = "abcdefghijklmnopqrstuvwxyz";
    pthread_t thread;
    pthread_attr_t attr;

    pthread_attr_init(&attr);
    pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);

    if (pthread_create(&thread, NULL, thread_routine, (void *)(buf))) {
            fprintf(stderr, "error creating a new thread \n");
            exit(1);
    }

    printf("Main is now exiting.\n");
    pthread_exit(NULL);
}
```