

Chapter 1 Introduction (Parallel Programming)

Wuu Yang

National Chiao-Tung University, Taiwan, R.O.C.

first draft: January 6, 2011

current version: July 5, 2011

©July 5, 2011 by Wuu Yang. All rights reserved.

Text: Lin and Snyder, Principles of Parallel Programming, 2009, Prentice Hall.

Outline

1. Introduction
2. Understand Parallel Computers
3. Reasoning About Performance
4. First Steps Toward Parallel Programming
5. Scalable Algorithmic Techniques
6. Programming With Threads
7. MPI and Other Local View Languages
8. ZPL and Other Global View Languages
9. Assessing the State of the Art
10. Direction in Parallel Programming

11. Writing Parallel Programs

Parallelism in Computer Programs

Existing programs are mostly written for a *sequential computer model*. They cannot improve performance much even if they run on a modern parallel computer.

In a sequential program, when an instruction is executed, it is assumed that *all* preceding instructions have finished execution. This assumption is so strong that sequential programming is much easier than parallel programming.

For a parallel program, the programmer (and the compiler) must carefully specify, for every instruction i , what instructions must be finished before instruction i can be initiated.

Innovations in modern computer architectures:

1. The architecture provides separate wires for data and instructions so that data and instructions can be referenced in parallel.

2. Instruction execution is pipelined.
3. The processor issues several instructions at a time.
4. Prefetch data and instructions speculatively.
5. Execute instructions speculatively.

start here

Concurrent Programming

Several activities occur at the same time is called *concurrency*.

Concurrency may occur at various levels. Hardware exception handlers, processes, and Unix signal handles are examples.

OS kernels can use concurrency to run several applications at the same time. Applications can also use concurrency to run several tasks of the applications. Application-level concurrency is useful in the following ways:

- Computing in parallel on multiprocessors. In a uniprocessor machine, concurrency is implemented with interleaved execution of different programs. In a multiprocessor machine, several applications truly run simultaneously. A *parallel application* that are partitioned into concurrent flows can sometimes run much faster on such multiprocessor machines.
- Accessing slow I/O devices. When an application is waiting for

data from a slow I/O device, the kernel can keep the CPU by running other processes. An application can also be designed so that overlapping computation with I/O activities.

- Interacting with humans. A user may want to perform several tasks at the same time. For instance, he may want to resize a window while printing a document.
- Reducing latency by deferring work. Applications can reduce the latency of some operations by deferring other operations and performing them concurrently.
- Servicing multiple network clients. A large server can service many clients at the same time by creating a separate logic flow for each client.

Applications that use application-level concurrency are called concurrent programs. OS provides three basic approaches for building concurrent programs:

- Processes. An application consists of several processes. Each process has a separate virtual address spaces. Processes use *interprocess communication* (IPC) to communicate.
- I/O multiplexing. An application is a single process which explicitly schedules its own logical flows. The main program jumps around the logical flows according to a state-machine model.
- Threads. Threads are logical flows in the context of a single process (i.e., they share the same address space). Similar to processes, threads are scheduled by the kernel.

§1 Parallel Hardware

§1.1 Processors

There are three approaches to parallel computation:

- Pipeline. Divide a task into multiple stages so that all stages can proceed on different parts of the data at the same time.
- Duplication. We may duplicate a piece of hardware functional units so that all units can work on different part of the data at the same time. Duplication is also called a *homogeneous system*.
- Team. A system consists of a team of heterogeneous functional units, each with a special expertise.
- A combination of pipeline, duplication, and team.

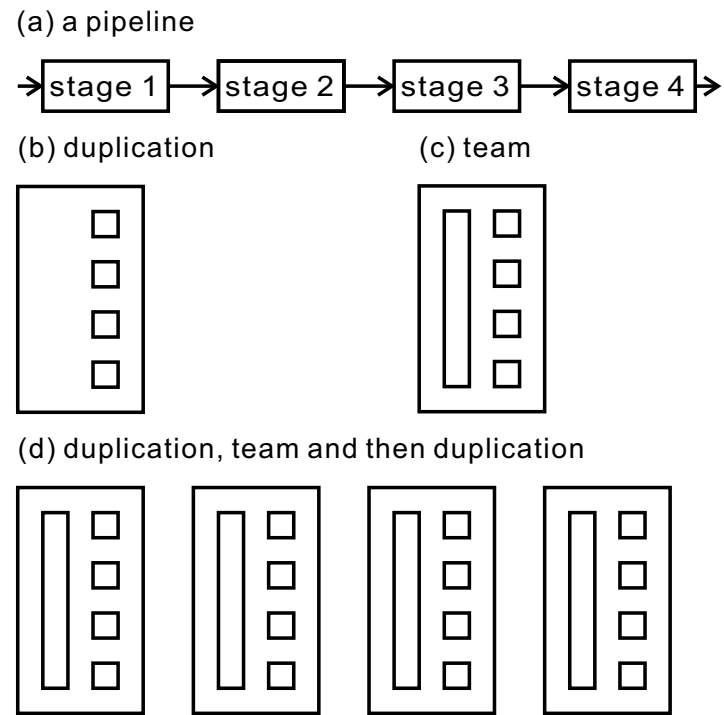


Figure 1: Parallel architecture.

Why Pipeline?

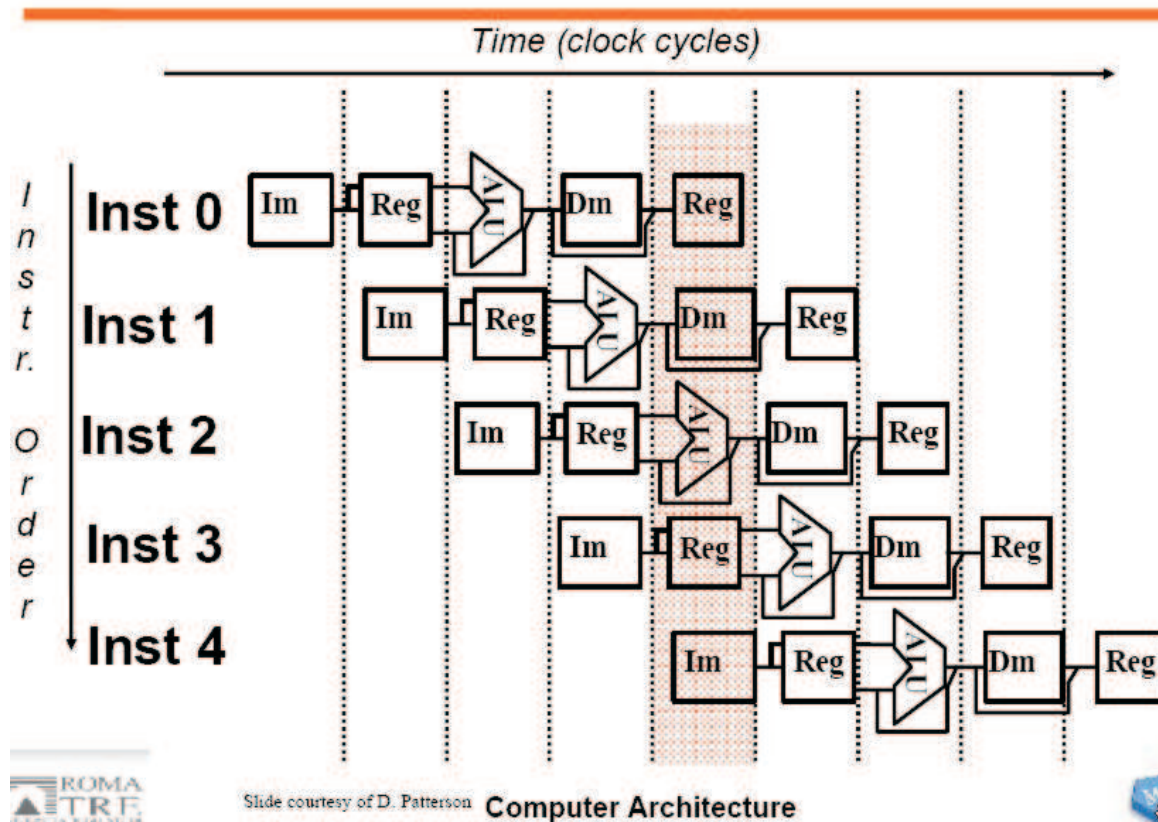


Figure 2: Pipeline in computer architecture.

Pipeline, duplication, and team may appear in several levels. For instance, the floating-point multiplication unit in ALU could be partitioned into 5 stages. Five multiplications could be conducted, at different stages, at the same time. Of course, to fully utilize the multiplication pipeline, data must be arranged properly so that no or little wait will be introduced. The input data will also need to be sizable so that there is significant overall speedup. A second example of a pipeline is the usual instruction processing in a CPU is divided into instruction fetch, instruction decode, instruction execution, and result storage. Pipeline can also occur at the CPU/computer level where one computer is responsible for registration, one computer is responsible for search, and one computer is responsible for database management.

Similarly, duplication may appear in several levels. There could be 3 multiplication units inside the ALU in a CPU. There could be multiple cores in a chip. There could also be multiple chips on a

mother board. A computer system could consist of many individual computers. Similar to a pipeline architecture, software must be provided in order to fully utilize the duplicated hardware. An Nvidia GPC consists of many identical processing elements.



Figure 3: Left is Nvidia GeForce and right is Tesla.

A team consists of several distinct units. For instance, the ALU could contain an adder, a multiplier and a logic unit. A computer could consist of a CPU and a GPU.

Pipeline, duplication, and team could be combined in various forms. There could be three multiplication pipelines. For example,

IBM Cell processors include 1 Power Processor Element (PPE) and multiple Synergistic Processing Elements (SPE). The SPEs is a duplication of several identical processors. A Cell processor is team of an SPE and a duplication of several PPEs.

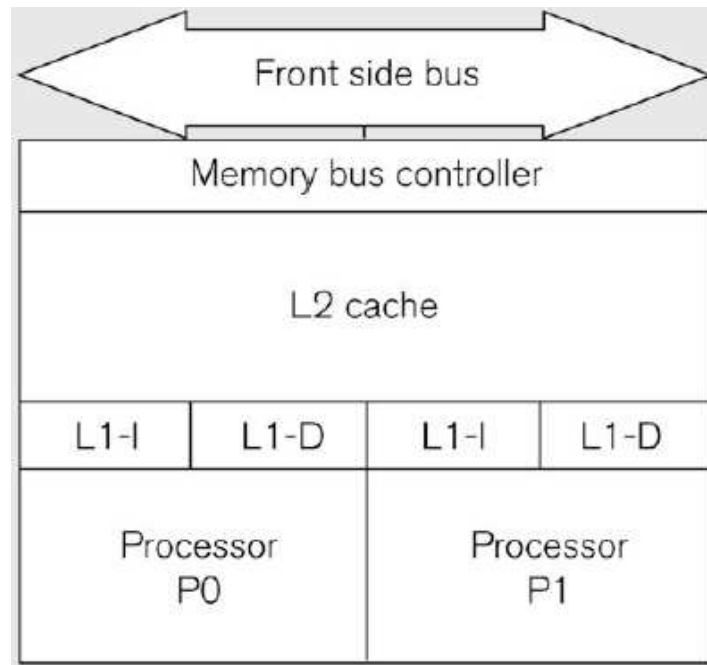
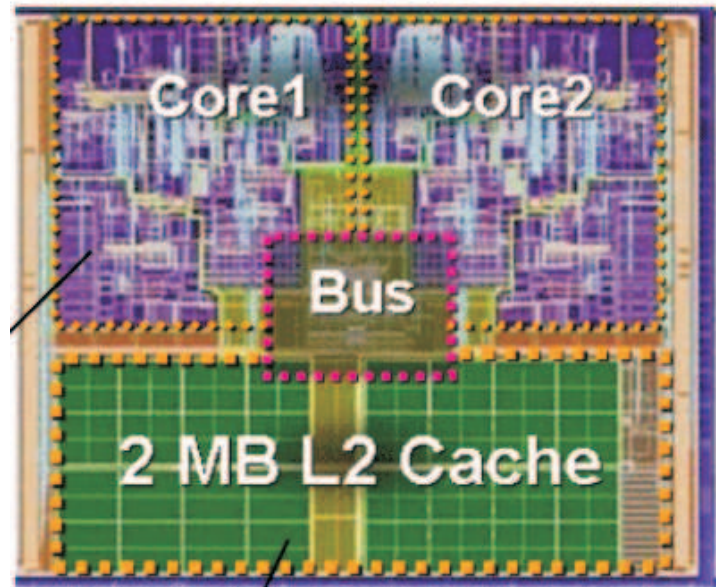


Figure 4: Intel Core Duo as a duplicate.



Source: Intel Corp.

Figure 5: Intel Core Duo as a duplicate.

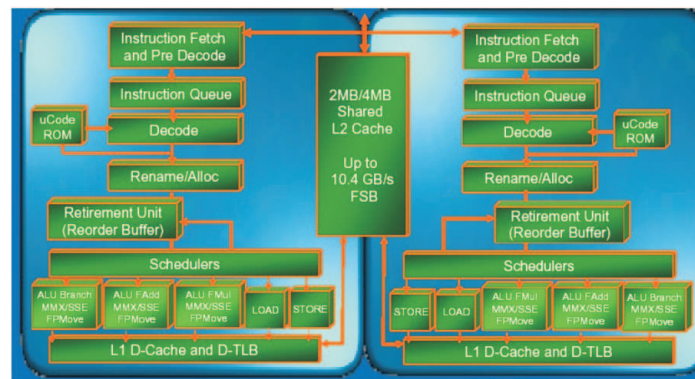


Figure 6: Intel Core Duo as a duplicate.

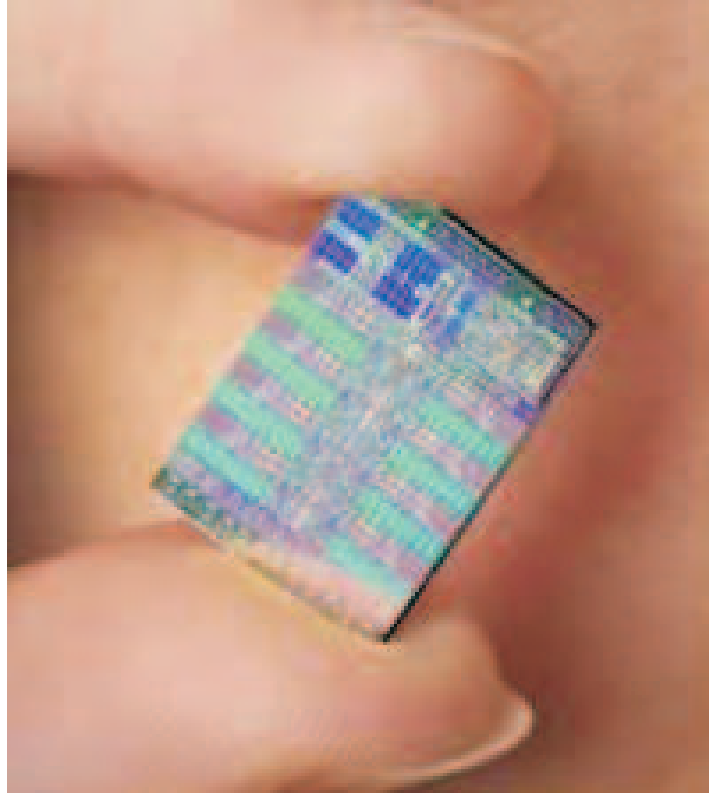


Figure 7: IBM Cell processor.

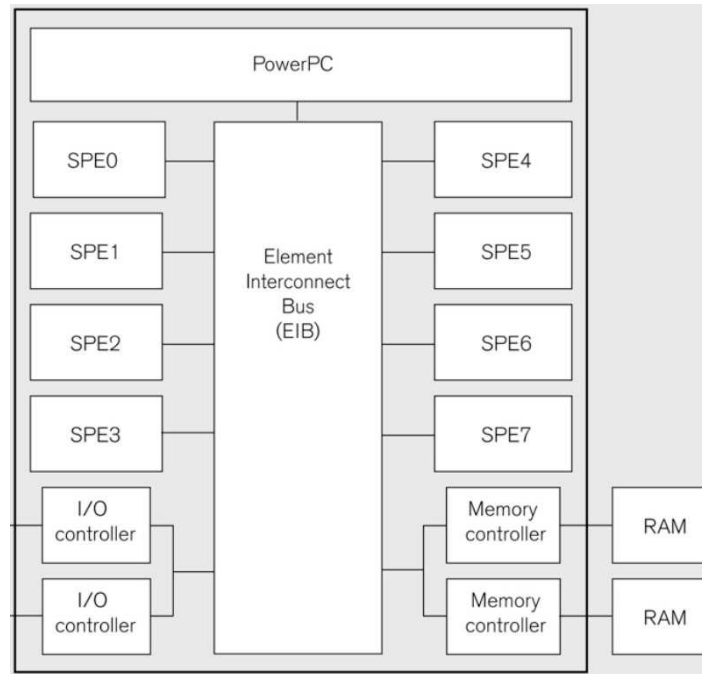


Figure 8: IBM Cell processor as a team.

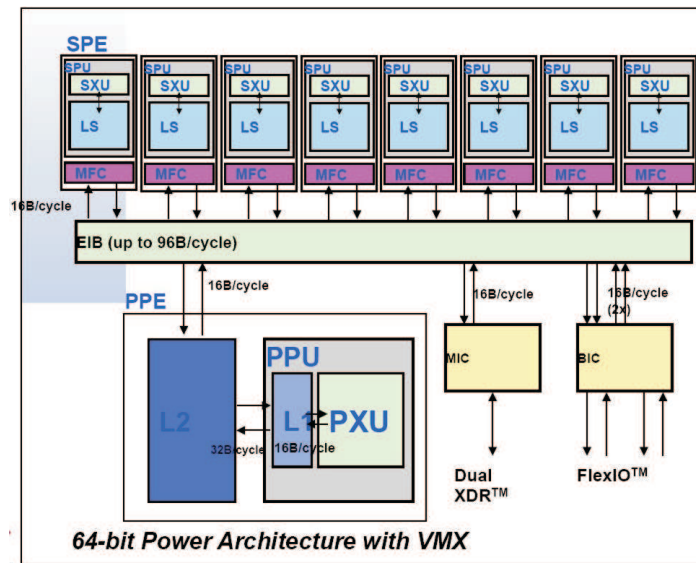


Figure 9: IBM Cell processor as a team.

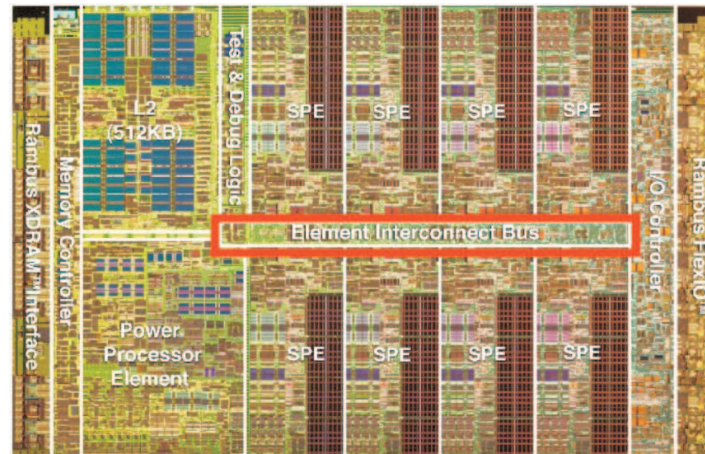


Figure 10: IBM Cell processor as a team.

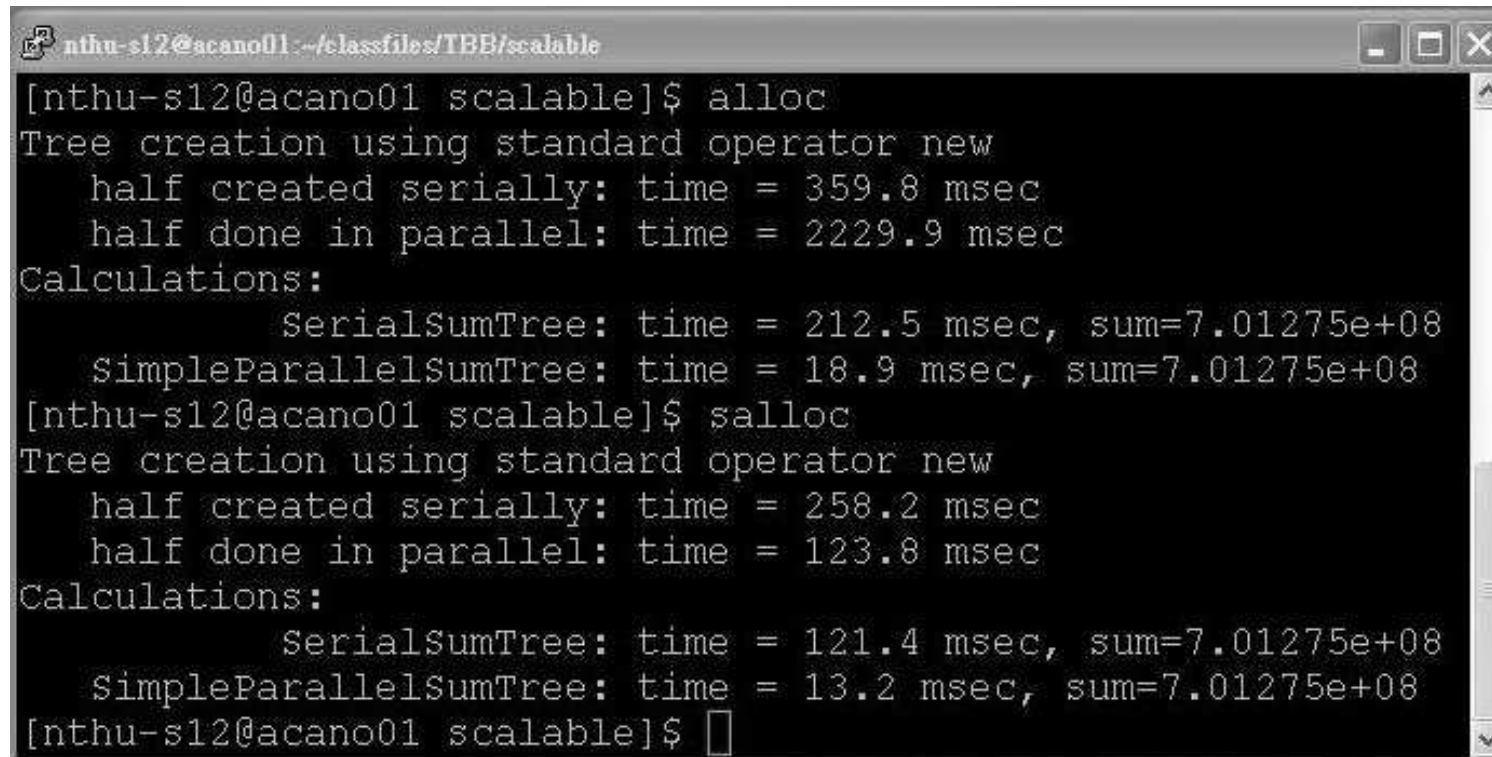
In addition to processor architecture, registers, memory hierarchy, cache, bus, interconnection network, i/o all influence the performance of a parallel system.

§1.2 Memory

In addition to functional units, memory is also an indispensable component. Logically, there is a chunk of memory which all processors can access. However, the organization of the memory structure significantly affects the *performance* of a parallel system. Memory could be

1. shared: all processors share the same memory. This will create the race problem and hence synchronization/mutual exclusion is needed.
2. distributed: Each processor has its own memory. When one processor needs another processor's data, it sends a request and waits for a response. In order to avoid repeated request/response, a processor may save a local copy for private use. This creates multiple copies of a data. These copies may be incoherent.

Example. Figure 11 shows a 10x speedup of a parallel program.



```
nthu-s12@acano01:~/classfiles/TBB/scalable
[nthu-s12@acano01 scalable]$ alloc
Tree creation using standard operator new
  half created serially: time = 359.8 msec
  half done in parallel: time = 2229.9 msec
Calculations:
  SerialSumTree: time = 212.5 msec, sum=7.01275e+08
  SimpleParallelSumTree: time = 18.9 msec, sum=7.01275e+08
[nthu-s12@acano01 scalable]$ salloc
Tree creation using standard operator new
  half created serially: time = 258.2 msec
  half done in parallel: time = 123.8 msec
Calculations:
  SerialSumTree: time = 121.4 msec, sum=7.01275e+08
  SimpleParallelSumTree: time = 13.2 msec, sum=7.01275e+08
[nthu-s12@acano01 scalable]$
```

Figure 11: Parallel architecture.

parallel programming languages

fully automatic

compiler/
translator/
transformer

parallel hardware

Figure 12: Parallel programming.

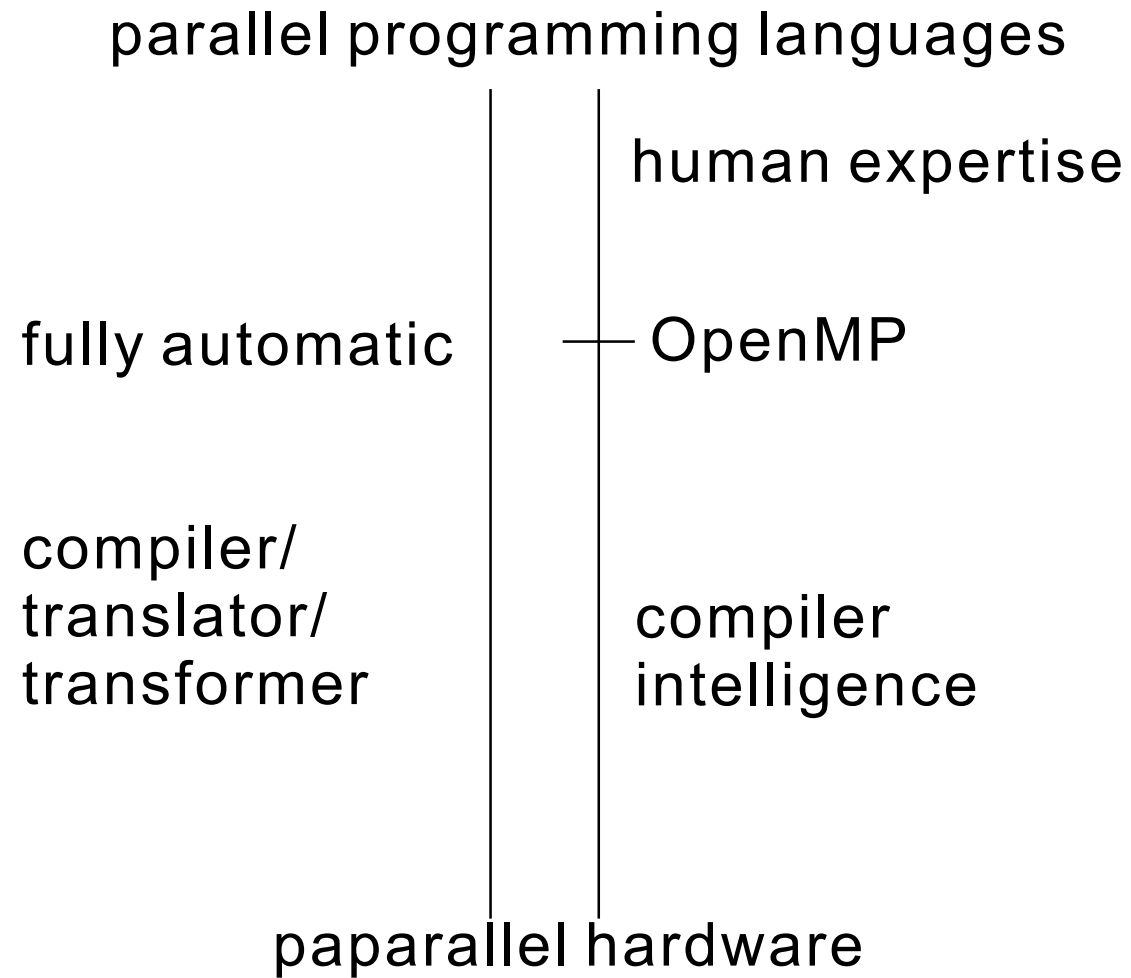


Figure 13: Parallel programming.

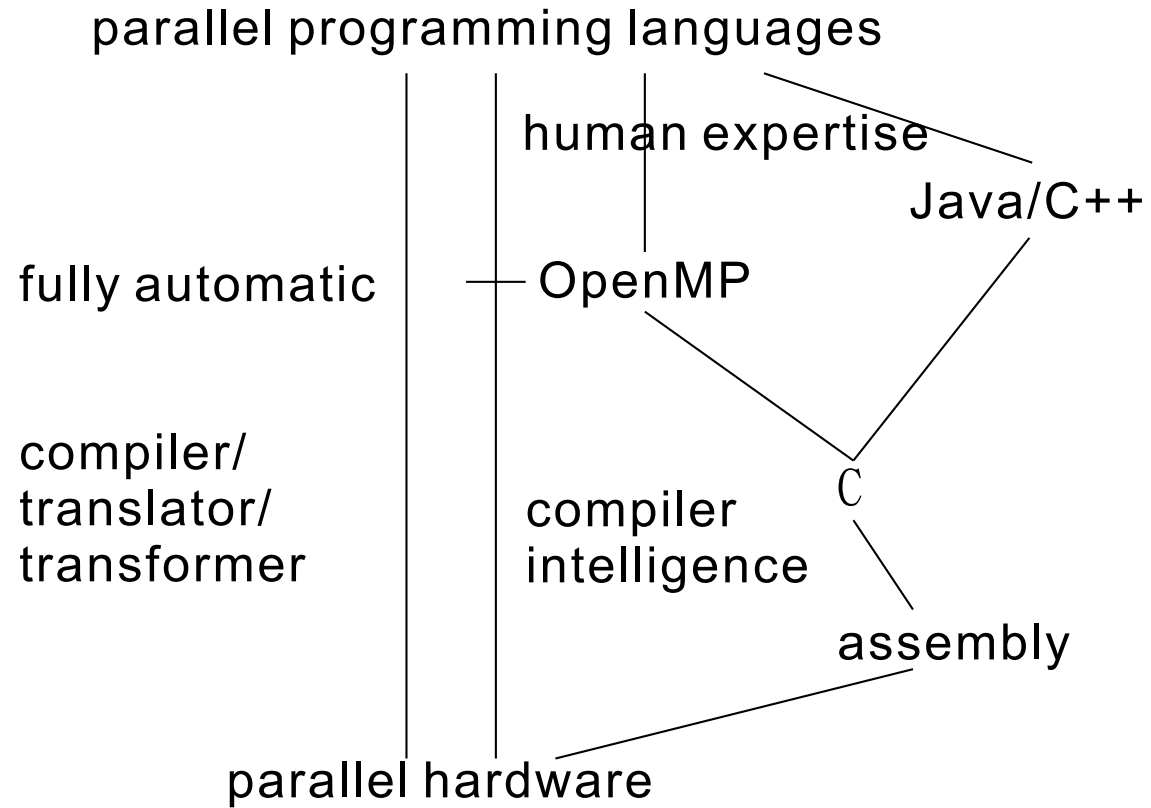


Figure 14: Parallel programming.

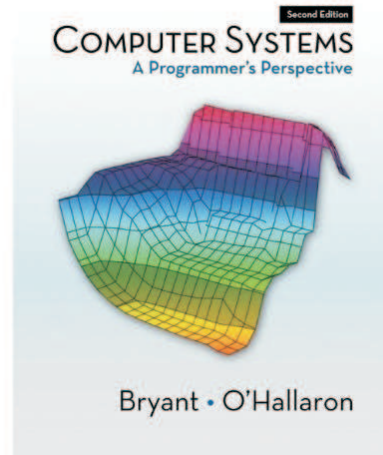
§2 Parallel Programming

In general, there are three types of programming supports for concurrent software.

1. The first type is through thread libraries. Examples are Win32 API and POSIX thread libraries. This type lets the programmers to manage their concurrent computing at the level of threads. For some applications, this level of concurrency abstraction can create a lot of burden to the programmers.
2. The second type is message passing system calls between processes. An example is MPI (message-passing interface). Each message passing may need some service from operating systems and can be quite costly.
3. In the last several years, a new type has emerged in the form of compiler directives. This type allows programmers to construct concurrent programs with high abstraction. It also utilizes

advanced techniques in compilation to foster succinct coding style and generate highly efficient object code. The most prominent one is OpenMP by Intel. As a result, concurrent programs developed with such a type of support can be small and easy to understand and maintain. OpenMP has now emerged as a de facto industry standard. More and more industry projects are now using OpenMP to write concurrent programs.

Text: R.E. Bryant and D.O'Hallaron, Computer Systems A Programmer's Perspective, Chapter 13.



See “Pren-
tice.Hall.Computer.Systems.A.Programmer’s.Perspective.pdf”,
chapter 11, pp. 563-604.
See also “comp322-lec10-f09-v1.pdf”, COMP 322: Principles of
Parallel Programming Lecture 10: POSIX Threads (Chapter 6)
Fall 2009 Vivek Sarkar Department of Computer Science Rice
University vsarkar@rice.edu

Outline

1. Concurrent Programming with Processes
2. Concurrent Programming with I/O Multiplexing
3. Concurrent Programming With Threads
4. Shared Variables in Threaded Programs
5. Synchronizing Threads With Semaphores
6. Putting It Together: A Concurrent Server Based on Prethreading
7. Other Concurrency Issues
8. Summary

§13.1 Concurrent Programming with Processes

We can build a concurrent programs with processes, using functions such as `fork`, `exec`, `waitpid`, etc. For instance, in a client/server system, the server could create a new process upon receiving a client connection request.

Example. Figure 13-1 — 13-5.

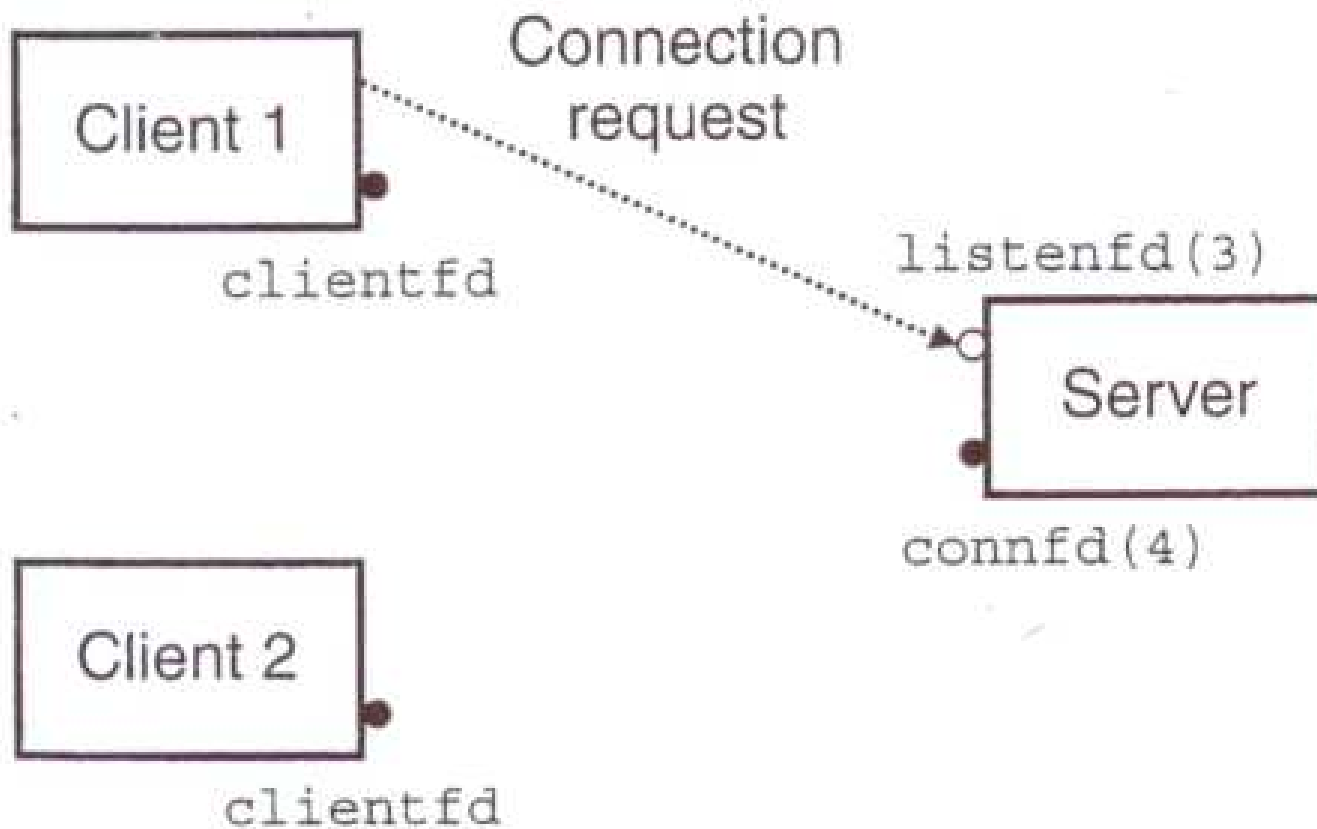


Figure 13.1 Step 1: Server accepts connection request from client.

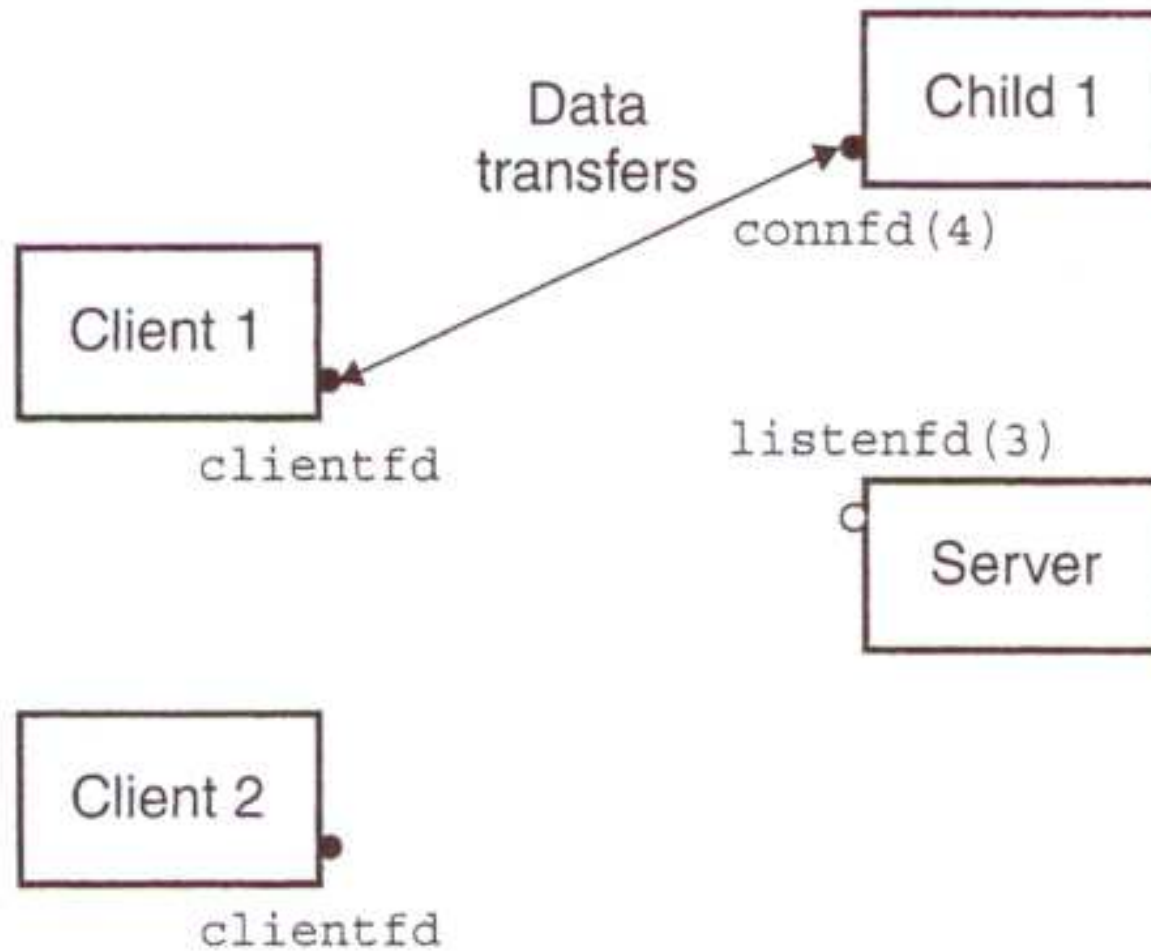


Figure 13.2 Step 2: Server forks a child process to service the client.

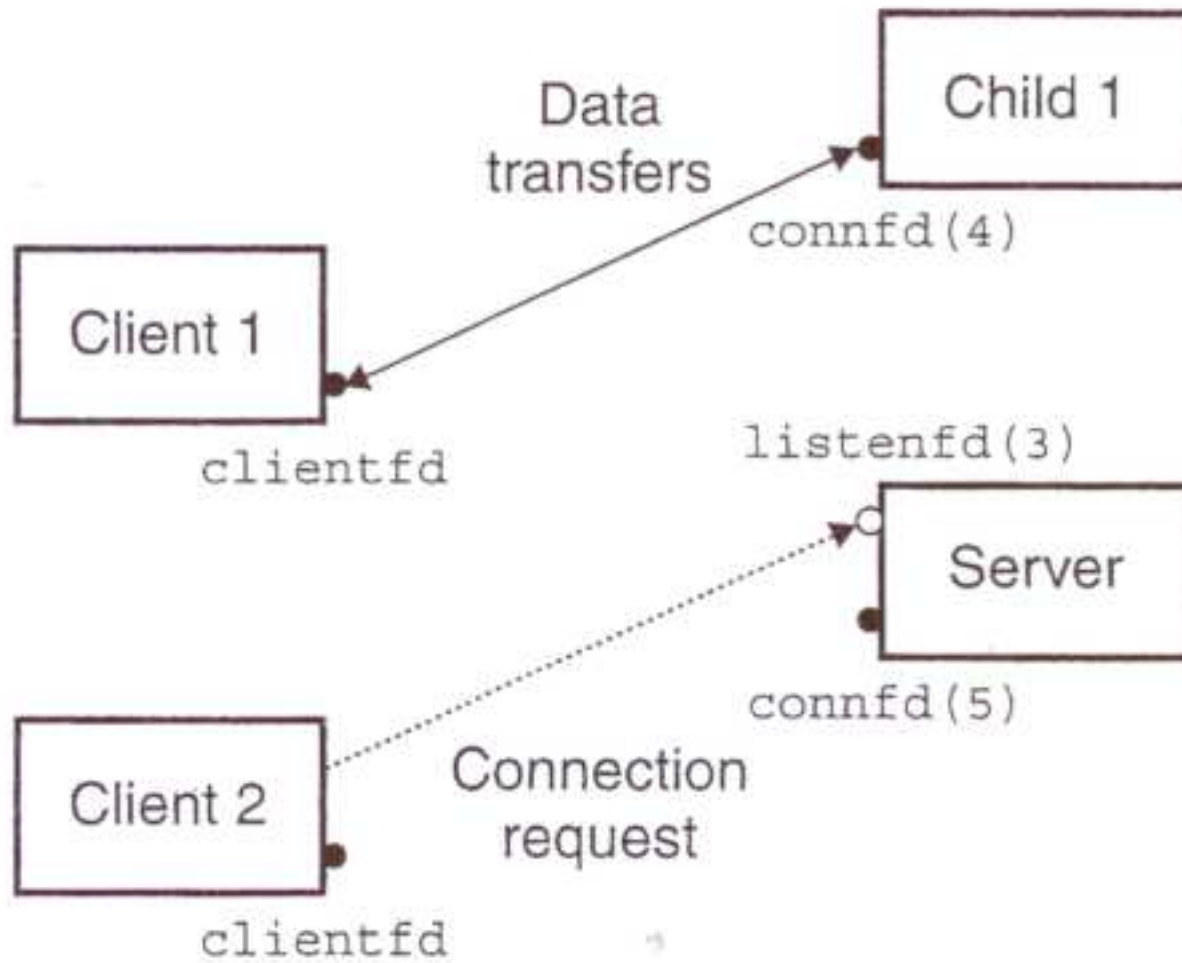


Figure 13.3 Step 3: Server accepts another connection request.

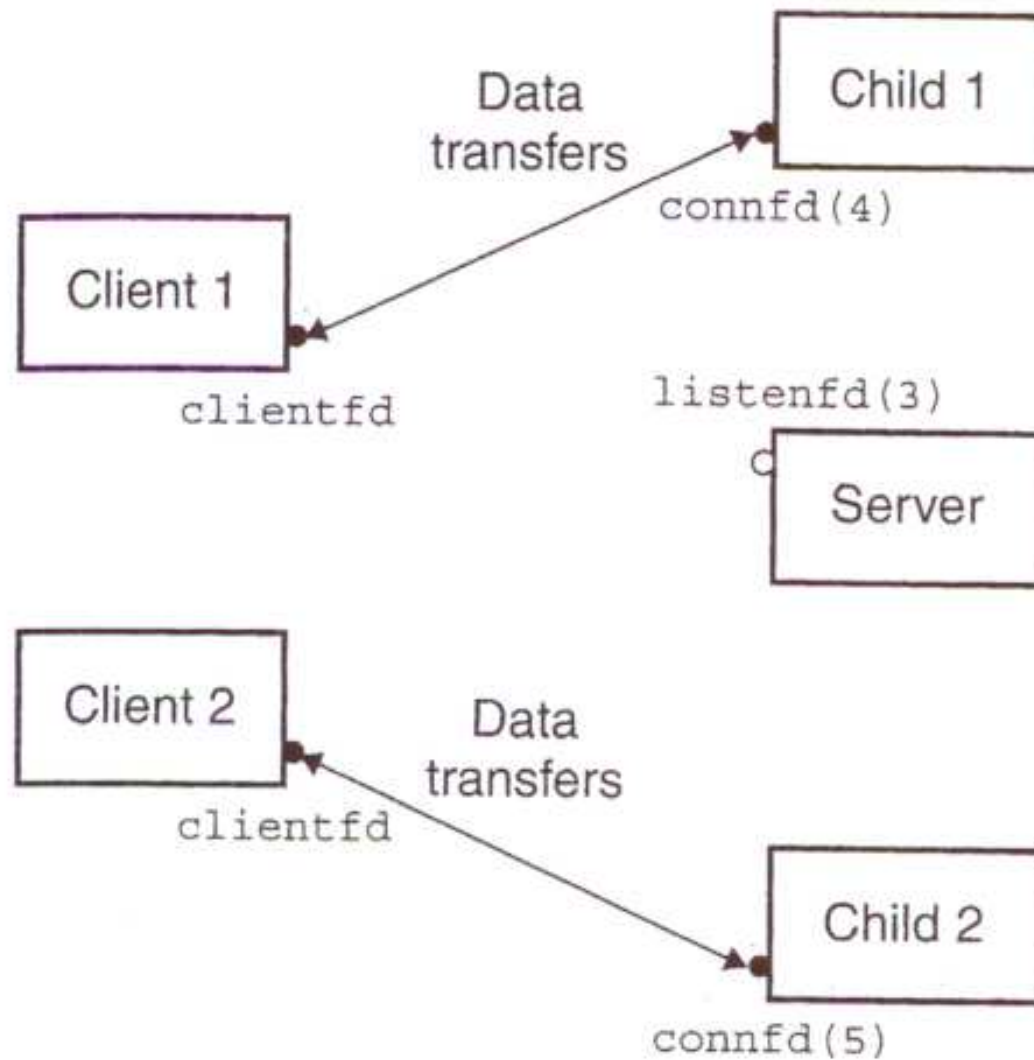


Figure 13.4 Step 4: Server forks another child to service the new client.

```

/** Figure 13.5 Concurrent echo server based on porcesses.
 * The parent forks a child to handle each new connection request.
 * code/conc/echoserverp.c
 */

#include "csapp.h" void echo(int connfd);

void sigchld_handler(int sig) {
    while (waitpid(-1, 0, WNOHANG) > 0)
        ;
    return;
}

int main(int argc, char **argv) {
    int listenfd, connfd, port, clienten=sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;

```

```

if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);
}
port = atoi(argv[1]);

Signal(SIGCHLD, sigchld_handler);
listenfd = Open_listenfd(port);
while (1) {
    connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    if (Fork() == 0) {
        Close(listenfd); /* Child closes its listening socket */
        echo(connfd);    /* Child services client */
        Close(connfd);  /* Child closes connection with client */
        exit(0);        /* Child exits */
    }
}

```

```
    Close(connfd); /* Parent closes connected socket (important!) *  
  }  
}
```

§13.2 Concurrent Programming with I/O Multiplexing

Suppose we want to write an echo server that could wait for two things:

1. a connection request from a network client
2. a command from a user on the keyboard

For the former, the server will wait for an `accept` call while for the latter, the server will wait for a `read` call. To wait for both events, we can use either. Instead we will use i/o multiplexing.

The basic idea is the `select` call, which asks the kernel to suspend the process. The process will be re-activated only if one or more i/o events have occurred, for example:

1. return when any descriptor^a in the set $\{0, 4\}$ is ready for reading;

^aA *descriptor* is represented as a small integer, such as 5, 7, etc.

2. return when any descriptor in the set $\{1, 2, 7\}$ is ready for writing;
3. timeout if 152.13 seconds have elapsed waiting for an i/o event to occur.

`Select` can wait for a set of descriptors, in addition to many other scenarios.

`Select` manipulates sets of type `fd_set`. A set of n descriptors can be considered as n bits, one for each descriptor. A bit could be turned on or off. For type `fd_set`, a programmer can

1. allocate them
2. assign one variable to another
3. modify and inspect with `FD_ZERO`, `FD_CLR`, `FD_SET`, and `FD_ISSET`.

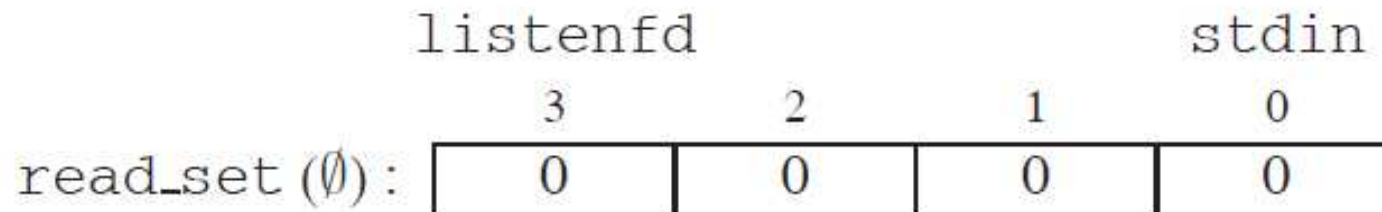
```
#include <unistd.h>
#include <sys/types.h>

int select(int n, fd_set *fdset, NULL,, NULL, NULL);
    // returns non-zero count of ready descriptors, -1 on error

FD_ZERO(fd_set *fdset); // clear all bits in fdset
FD_CLR(int fd, fd_set *fdset); // clear bit fd in fdset
FD_SET(int fd, fd_set *fdset); // turn on bit fd in fdset
FD_ISSET(int fd, fd_set *fdset); // is bit fd in fdset turned on?
```

Figure (p. 854) Macros for manipulating descriptor sets.

The `select` function blocks until at least one descriptor in the read set is ready for reading. A descriptor k is ready for reading if and only if a request to read one byte from that descriptor would not block. As a side effect, `select` modifies the `fd_set` pointed by the argument `fdset` to indicate a subset of the read set, called the *ready set*, consisting of the descriptors in the read set that are ready for reading. The value returned from the `select` call is the size of the ready set.



```
/** Figure 13.6 An echo server that uses I/O multiplexing.
 *     - The server uses select to wait for connection requests
 *     on a listening descriptor and commands on standard input.
 *     code/conc/select.c
 */
```

```
#include "csapp.h" void echo(int connfd); void command(void);
```

```
int main(int argc, char **argv) {
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    fd_set read_set, ready_set;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
```

```

        exit(0);
    }
    port = atoi(argv[1]);
    listenfd = Open_listenfd(port);

    FD_ZERO(&read_set);          /* Clear read set */
    FD_SET(STDIN_FILENO, &read_set); /* Add stdin to read set */
    FD_SET(listenfd, &read_set);    /* Add listenfd to read set */

    while (1) {
        ready_set = read_set;
        Select(listenfd+1, &ready_set, NULL, NULL, NULL);
        if (FD_ISSET(STDIN_FILENO, &ready_set))
            command(); /* Read command line from stdin */
        if (FD_ISSET(listenfd, &ready_set)) {
            connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);

```

```
        echo(connfd); /* Echo client input until EOF */
        Close(connfd);
    }
}
}
```

```
void command(void) {
    char buf[MAXLINE];
    if (!Fgets(buf, MAXLINE, stdin))
        exit(0); /* EOF */
    printf("%s", buf); /* Process the input command */
}
```

	listenfd			stdin
	3	2	1	0
read_set ({0,3}):	1	0	0	1

	listenfd			stdin
	3	2	1	0
ready_set ({0}):	0	0	0	1

`open_listenfd` opens a listening descriptor (which is 3).

`FD_ZERO` creates an empty read set. Then `FD_SET` sets up a read set (`{0, 3}`) with descriptors 0 (standard input) and 3 (the listening descriptor).

In line 24, we call the `select` function, which blocks until either the listening descriptor or the standard input is ready for reading.

Once `select` call returns, we use `FD_ISSET` macro to check which is ready for reading.

If the standard input is ready, we call `command` to read a line from the standard input.

If the listening descriptor is ready, we call `accept` to obtain a connected descriptor and then call the `echo` function.

We might want to modify the above program so that it echoes one text line each time through the server loop.

§13.2.1 A Concurrent Event-Driven Server Based on I/O Multiplexing

The `select` function implements an *event-driven* model: Its actions depend on the incoming events (be it from the standard input or from a listening descriptor). This model can be described with a *finite state machine*. The machine is in a state (called the current state). Upon receiving an event, it transits to another state. During the transition, the machine will perform some action.

In Figure 13.7, state k_k means “waiting for descriptor d_k to be ready for reading”, its input event is “descriptor d_k becomes ready”, and the transition means “read one input text line from d_k ”.

The server uses `select` to detect the occurrence of an event.

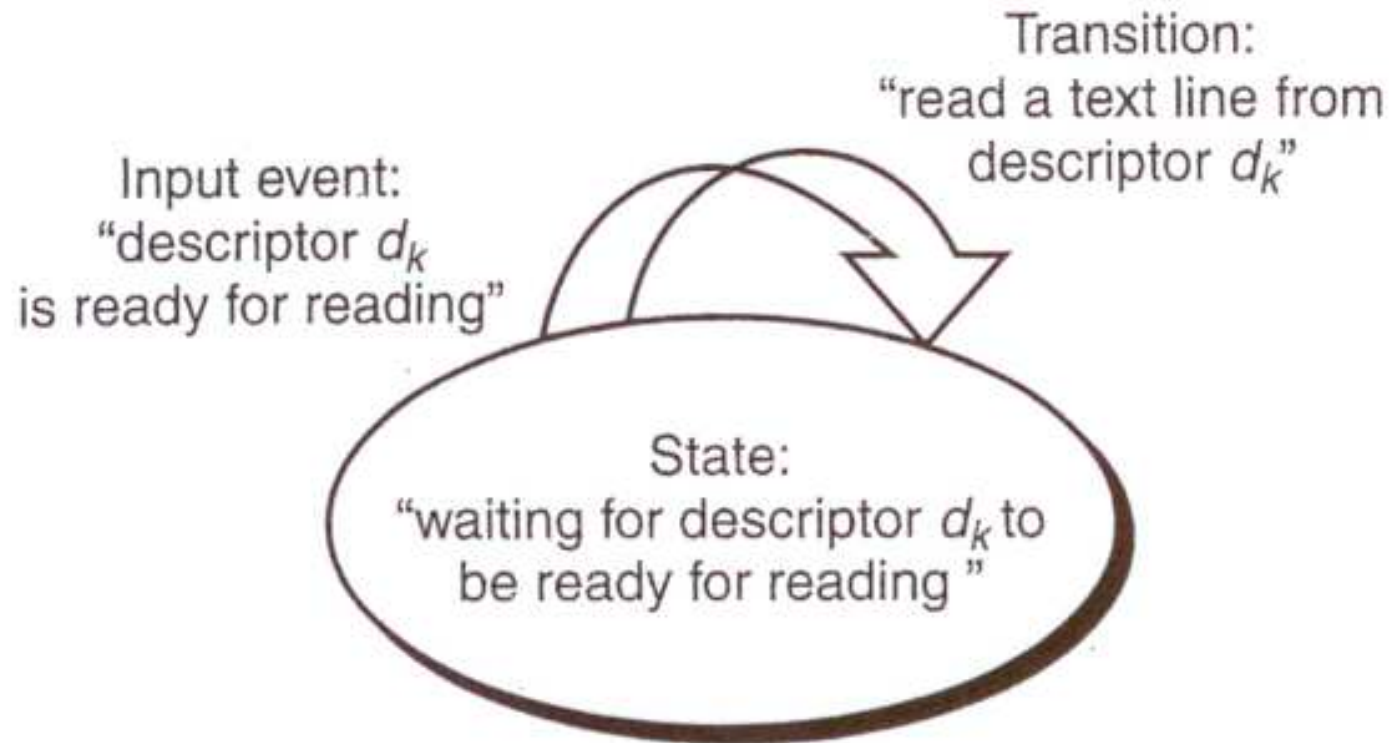


Figure 13.7 State machine for a logical flow in a concurrent event-driven echo server.

```
/** Figure 13.8 Concurrent echo server based on I/O multiplexing.
 *     - Each server iteration echoes a text line from each ready
 *     descriptor.
 *     code/conc/echoservers.c
 */
```

```
#include "csapp.h"
```

```
typedef struct { /* represents a pool of connected descriptors */
    int maxfd;          /* largest descriptor in read_set */
    fd_set read_set;    /* set of all active descriptors */
    fd_set ready_set;  /* subset of descriptors ready for reading */
    int nready;        /* number of ready descriptors from select */
    int maxi;         /* highwater index into client array */
    int clientfd[FD_SETSIZE]; /* set of active descriptors */
    rio_t clientrio[FD_SETSIZE]; /* set of active read buffers */
};
```

```
} pool;

int byte_cnt = 0; /* counts total bytes received by server */

int main(int argc, char **argv) {
    int listenfd, connfd, port;
    socklen_t clientlen = sizeof(struct sockaddr_in);
    struct sockaddr_in clientaddr;
    static pool pool;

    if (argc != 2) {
        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);
```

```

listenfd = Open_listenfd(port);
init_pool(listenfd, &pool);
while (1) {
    /* Wait for listening/connected descriptor(s) to become ready
    pool.ready_set = pool.read_set;
    pool.nready = Select(pool.maxfd+1, &pool.ready_set,
                        NULL, NULL, NULL);

    /* If listening descriptor ready, add new client to pool */
    if (FD_ISSET(listenfd, &pool.ready_set)) {
        connfd = Accept(listenfd, (SA *)&clientaddr, &clientlen);
        add_client(connfd, &pool);
    }

    /* Echo a text line from each ready connected descriptor */
    check_clients(&pool);
}

```

}
}

```

/** Figure 13.9 init_pool: Initializes the pool of active clients.
 *     code/conc/echoservers.c
 */

void init_pool(int listenfd, pool *p) {
    /* Initially, there are no connected descriptors */
    int i;
    p->maxi = -1;
    for (i=0; i< FD_SETSIZE; i++)
        p->clientfd[i] = -1;

    /* Initially, listenfd is only member of select read set */
    p->maxfd = listenfd;
    FD_ZERO(&p->read_set);
    FD_SET(listenfd, &p->read_set);
}

```

In Figure 13.9, `init_pool` initializes the client pool. The `clientfd` array represents a set of connected descriptors, with `-1` denoting an available slot. Initially, the `clientfd` array is vacant and the listening descriptor is the only descriptor in the `select` read set.

```
/** Figure 13.10 add_client: Adds a new client connection to the pool
 *     code/conc/echoservers.c
 */
```

```
void add_client(int connfd, pool *p) {
    int i;
    p->nready--;
    for (i = 0; i < FD_SETSIZE; i++) /* Find an available slot */
        if (p->clientfd[i] < 0) {
            /* Add connected descriptor to the pool */
            p->clientfd[i] = connfd;
            Rio_readinitb(&p->clientrio[i], connfd);

            /* Add the descriptor to descriptor set */
            FD_SET(connfd, &p->read_set);
        }
}
```

```
        /* Update max descriptor and pool highwater mark */
        if (connfd > p->maxfd)
            p->maxfd = connfd;
        if (i > p->maxi)
            p->maxi = i;
        break;
    }
if (i == FD_SETSIZE) /* Couldn't find an empty slot */
    app_error("add_client error: Too many clients");
}
```

In Figure 13.10, `add_client` adds a new client to the pool of active clients to a vacant slot in the `clientfd` array. The connected descriptor is entered into a vacant slot in the `clientfd` array and a corresponding RIO read buffer is initialized. Later we can call `Rio_readlineb` on the descriptor.

The `maxfd` variable keeps track of the largest file descriptor for `select`. The `maxi` variable keeps track of the number of file descriptors in the `clientfd` array.

```

/** Figure 13.11 check_clients: Services ready client connections.
 *     code/conc/echoservers.c
 */

void check_clients(pool *p) {
    int i, connfd, n;
    char buf[MAXLINE];
    rio_t rio;

    for (i = 0; (i <= p->maxi) && (p->nready > 0); i++) {
        connfd = p->clientfd[i];
        rio = p->clientrio[i];

        /* If the descriptor is ready, echo a text line from it */
        if ((connfd > 0) && (FD_ISSET(connfd, &p->ready_set))) {
            p->nready--;

```

```

if ((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    byte_cnt += n;
    printf("Server received %d (%d total) bytes on fd %d\n",
        n, byte_cnt, connfd);
    Rio_writen(connfd, buf, n);
}

/* EOF detected, remove descriptor from pool */
else {
    Close(connfd);
    FD_CLR(connfd, &p->read_set);
    p->clientfd[i] = -1;
}
}
}
}

```

`check_clients` echoes a text line from each ready connected descriptor.

In the `main` function (Figure 13.8), a listening fd is opened. The `main` function then calls `select` to wait for connection requests. When an event occurs, it checks if the event is a new connection request. If so, it calls `accept` to identify the new client, create a new connection fd, and add it to the pool (with `add_client`). Finally, it calls `check_client` to echo the input text line from each active client.

§13.2.2 Pros and Cons of I/O Multiplexing

Advantages and disadvantages:

1. An event-driven server based on I/O multiplexing runs in the context of a single process. This makes sharing data among flows easy.
2. Debugging is easier because it is a sequential program.
3. No process context switch is need. Hence it is more efficient.
4. Coding complexity is higher.
5. As long as a logical flow is reading a text line, no other logical flow can make progress.

In Unix, `ctrl-D` means end-of-file.

§13.3 Concurrent Programming With Threads

Threads combines the advantages of processes and data sharing.

A thread is a logical flow that runs in the context of a single process. There could be multiple threads running concurrently in a single process. Threads are scheduled automatically by the kernel.

Each thread has its own *thread context*, including

1. a thread id (TID, which is an integer)
2. a stack pointer
3. a program counter
4. general-purpose registers
5. condition codes

All threads in the same process share the entire virtual address space of the process. This means all data are shared among all threads in the same process. Sharing means that communication

among threads is very cheap. Specifically, the threads share

1. code
2. data
3. heap
4. shared libraries
5. open files

§13.3.1 Thread Execution Model

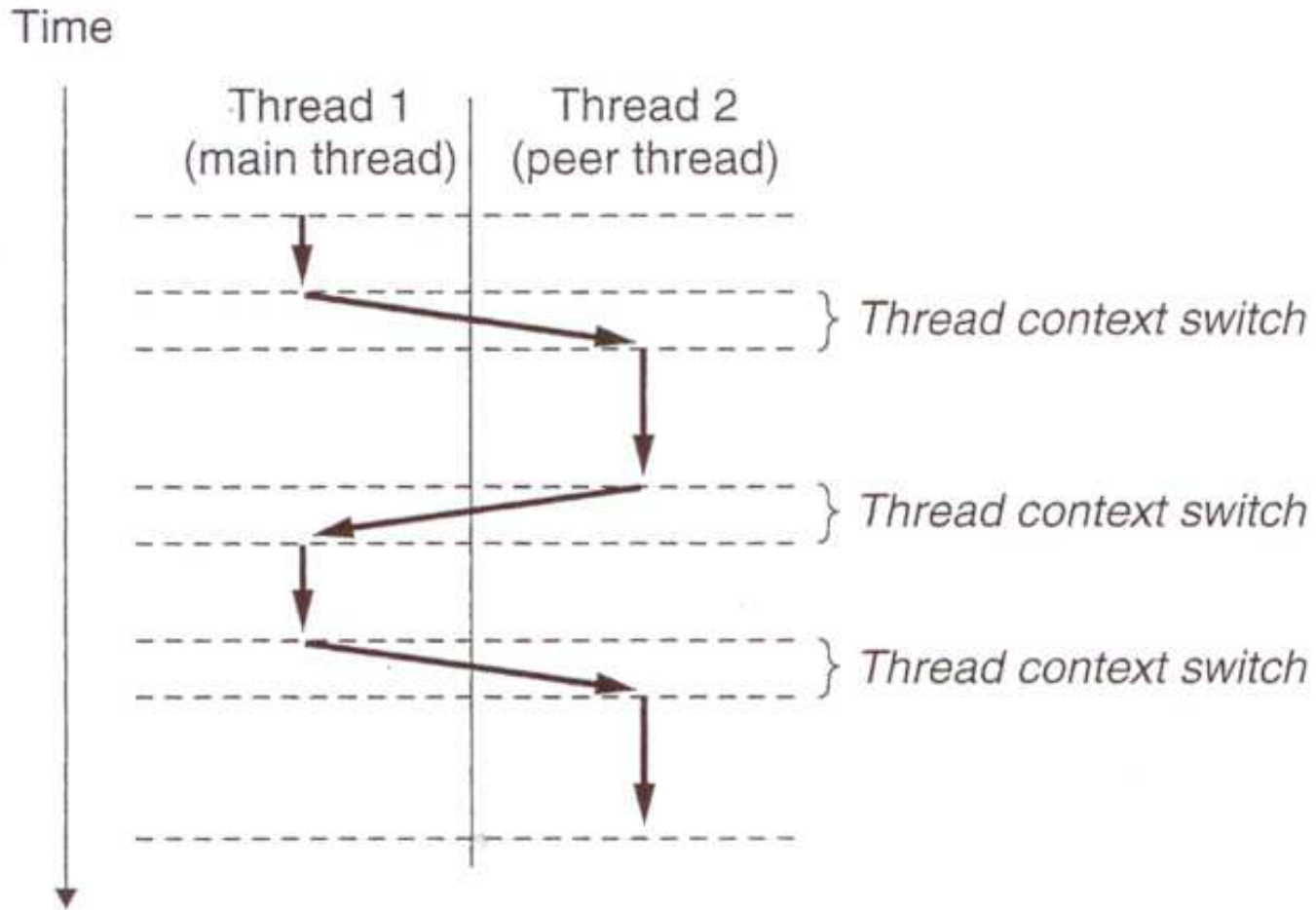


Figure 13.12 Concurrent thread execution.

Each process starts as a single thread (the main thread). The main thread may create other *peer threads*. All threads run concurrently. Control may pass to another thread via a context switch because a thread executes a slow system call (such as `read` and `sleep`) or it is interrupted by the system's interval timer.

Because a thread context is much smaller than a process context, thread switch is much faster than process switch. Unlike processes, threads are not organized in a rigid parent-child hierarchy. Instead, all threads are equal in a pool of *peers*. Thus, a thread can kill any of its peers or wait for any of its peers to terminate. Further, each peer can read and write the same shared data.

§13.3.2 Posix Threads

Posix threads (Pthreads) is a standard interface for manipulating threads from C programs. It was adopted in 1995 and is available on most Unix systems. Pthread defines about 60 functions that allow programs to create, kill, and reap threads, to share data safely with peer threads, and to notify peers about changes in the system state.

Figure 13.13 is a small Pthread program. `pthread_create` creates a new thread and gives it a new TID. `pthread_join` waits for the new thread to terminate. The newly created thread will execute the `thread` function.

```
/** Figure 13.13 hello.c: The Pthreads "Hello, world!" program.
 *   code/conc/hello.c
 */
```

```
#include "csapp.h" void *thread(void *vargp);

int main() {
    pthread_t tid;
    Pthread_create(&tid, NULL, thread, NULL);
    Pthread_join(tid, NULL);
    exit(0);
}

void *thread(void *vargp) /* thread routine */ {
    printf("Hello, world!\n");
    return NULL;
}
```

When creating a new thread, we may pass several parameters:

- a pointer to a pthread id structure.
- a pointer to a routine. The newly created thread will execute that thread.
- a pointer to the parameter of the thread routine. If there are several parameters, put them in a structure and pass a pointer to that structure.

The thread routine may return a result, which is a pointer to a structure.

The main thread waits for the new thread to terminate with `pthread_join`. Finally, when the main thread executes `exit`, all threads terminate.

§13.3.3 Creating Threads

```
#include <pthread.h>
typedef void *(func)(void *);
int pthread_create(pthread_t *tid, pthread_attr_t *attr,
                  func *f, void *arg);
```

The `attr` argument changes the default attributes of the newly created thread. When `pthread_create` returns, `tid` contains the id of the new thread. A thread can determine its id with `pthread_self`:

```
#include <pthread.h>

pthread_t pthread_self(void); // return thread id of caller
```

§13.3.4 Terminating Threads

A thread may terminate in several ways:

1. A thread terminates silently when its top-level thread routine returns.
2. A thread may terminate by call `pthread_exit`, which returns a pointer to the return value `thread_return`.
3. If the main thread calls `pthread_exit`, it waits for all other peer threads to terminate, and then terminates the main thread and the entire process with a return value `thread_return`.
4. Some peer thread calls Unix `exit` function, which terminates the process and all threads in the process.
5. Another thread terminates the current thread by calling the `pthread_cancel` function with the id of the current thread.

```
#include <pthread.h>
int pthread_exit(void *thread_return);
int pthread_cancel(pthread_t tid);
    // return 0 if ok, nonzero on error
```

Example. This example is taken from Lecture 10, an example of pthread, computing pi.

```
#include <pthread.h>
#include <stdlib.h>
#define NUM_THREADS 32
void *compute_pi (void *);
...
int main(...) {
    ...
    pthread_t p_threads[NUM_THREADS];
    pthread_attr_t attr;
    pthread_attr_init(&attr);
    for (i=0; i< NUM_THREADS; i++) {
        hits[i] = i;
        pthread_create(&p_threads[i], &attr, compute_pi,
                      (void*) &hits[i]);
    }
}
```

```

    }
    for (i=0; i< NUM_THREADS; i++) {
        pthread_join(p_threads[i], NULL);
        total_hits += hits[i];
    }
    ...
}

void *compute_pi (void *s) {
    int seed, i, *hit_pointer;
    double x_coord, y_coord;
    int local_hits;
    hit_pointer = (int *) s;
    seed = *hit_pointer;
    local_hits = 0;
    for (i = 0; i < sample_points_per_thread; i++) {

```

```
x_coord = (double)(rand_r(&seed))/((1<<15)-1) - 0.5;
y_coord =(double)(rand_r(&seed))/((1<<15)-1) - 0.5;
if ((x_coord * x_coord + y_coord * y_coord) < 0.25)
    local_hits++;
}
*hit_pointer = local_hits;
pthread_exit(0);
}
```

§13.3.5 Reaping Terminated Threads

Threads wait for other threads to terminate by `pthread_join`. This function blocks until the thread with `tid` terminates, assigns the `(void *)` pointer returned by the thread routine to the location pointed to by `thread_return`, and then reaps any memory resources held by the terminated thread.

Unlike Unix `wait`, this `pthread_join` can only wait for a specific thread to terminate. If the thread wishes to wait for an arbitrary set of threads to terminate, it has to use other less intuitive mechanisms.

```
#include <pthread.h>
int pthread_join(pthread_t tid, void **thread_return);
    // return 0 if ok, nonzero on error
```

§13.3.36 Detaching Threads

A thread may be *joinable* or *detached*. It is joinable immediately after creation and could become detached by the *pthread_detach* function.

```
#include <pthread.h>
int pthread_detach(pthread_t tid, void **thread_return);
    // return 0 if ok, nonzero on error
```

When a joinable thread exits or is killed by another thread, its memory is not freed until it is reaped by another thread. In contrast, a detached thread cannot be reaped or killed by other threads. Its memory resources are freed automatically by the system when it terminates.

It is preferred to use detached threads. For example, a server could create a new peer thread for each incoming connection request.

Since each connection is handled independently, it is unnecessary to ask the server to wait for the peer threads to terminate. In this case, a peer thread should detach itself before it begins processing the request.

§13.3.7 Initializing Threads

The `pthread_once` function allows you to initialize the state associated with a thread routine. The first time `pthread_once` is called with argument `once_control` the `init_routine` is invoked. Subsequent calls will do nothing. This is useful to initialize global variables shared by multiple threads.

```
#include <pthread.h>
pthread_once_t *once_control = PTHREAD_ONCE_INIT;
int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));
// always returns 0
```

§13.3.8 A Concurrent Server Based on Threads

```
/** Figure 13.14 Concurrent echo server based on threads.  
 * code/conc/echoserver.c  
 */
```

```
#include "csapp.h"
```

```
void echo(int connfd); void *thread(void *vargp);
```

```
int main(int argc, char **argv) {  
    int listenfd, *connfdp, port;  
    socklen_t clientlen=sizeof(struct sockaddr_in);  
    struct sockaddr_in clientaddr;  
    pthread_t tid;  
  
    if (argc != 2) {
```

```

        fprintf(stderr, "usage: %s <port>\n", argv[0]);
        exit(0);
    }
    port = atoi(argv[1]);

    listenfd = Open_listenfd(port);
    while (1) {
        connfdp = Malloc(sizeof(int));
        *connfdp = Accept(listenfd, (SA *) &clientaddr, &clientlen);
        Pthread_create(&tid, NULL, thread, connfdp);
    }
}

/* thread routine */ void *thread(void *vargp) {
    int connfd = *((int *)vargp);
    Pthread_detach(pthread_self());
}

```

```
    Free(vargp);  
    echo(connfd);  
    Close(connfd);  
    return NULL;  
}
```

Figure 13.14 is a concurrent echo server based on threads. The main thread waits for a connection request and then creates a peer thread to handle the request.

First the main thread needs to pass the connected descriptor to the newly created peer thread. However, the following code will cause a *race*:

```
connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);  
pthread_create(&tid, NULL, thread, &connfd);
```

```
void *thread(void *vargp) {  
    int connfd = *((int *) vargp);  
    . . .  
}
```

The assignment in the `thread` function (in the peer thread) may race with the `accept` for the next connection request (in the main thread). This is because the main and the peer threads work on the same descriptor. In order to avoid the race, the main thread allocates a new connection descriptor for each connection request.

A second issue is the memory leak. Since the main thread does not explicitly reap peer threads, each peer thread must be detached so that its memory resources will be reclaimed when it terminates (line 30).

The peer thread also needs to free the memory space allocated by

the main thread since the main thread will not free it (line 31).

§13.4 Shared Variables in Threaded Programs

An advantage of threads is that they share the same program variables. This makes programming much easier.

The memory issues in a program include

1. What is the underlying memory model for threads?
2. Under this model, how are the instances of the variables are mapped to memory?
3. How many threads reference each of the variables?

A variable is *shared* if multiple threads can reference the same instance of that variable.

```
/** Figure 13.15 Example program that illustrates different
 *     aspects of sharing.
 *     code/conc/sharing.c
 */
```

```
#include "csapp.h" #define N 2 void *thread(void *vargp);

char **ptr; /* global variable */

int main() {
    int i;
    pthread_t tid;
    char *msgs[N] = {
        "Hello from foo",
        "Hello from bar"
    };

    ptr = msgs;
    for (i = 0; i < N; i++)
        Pthread_create(&tid, NULL, thread, (void *)i);
    Pthread_exit(NULL);
}
```

```
}
```

```
void *thread(void *vargp) {  
    int myid = (int)vargp;  
    static int cnt = 0;  
    printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);  
    return NULL;  
}
```

In Figure 13.15, the main thread creates two peer threads. The main thread passes a unique id to each peer. The peer threads print their id along with a count of the total number of times that the thread routine has been invoked.

§13.4.1 Threads Memory Model

Each thread in a pool has its own separate *thread context*, which includes a thread ID, stack, stack pointer, program counter, condition code, and general-purpose registers.

All threads share the rest of the process context, which includes the entire virtual address space (read-only code, read/write data, the heap, and any shared library code and data area). The threads also share the same open files.

If a thread modifies a memory location, all other threads will see the change.

Each thread has a private stack that other threads cannot access. All these private stacks are in the stack area of the shared virtual address space. If a thread somehow obtains a pointer to another thread's stack, it can access that thread's private stack. In figure 13.15 line 26, a peer thread accesses the main thread's stack

through the global `ptr` variable.

§13.4.2 Mapping Variables to Memory

In a C program, there are three storage classes:

1. *Global variables.* Variables declared outside any functions are global variables. There is exactly one instance and it is referenced by all threads.
2. *Local automatic variables.* Variables declared inside a function without the `static` keyword is a local automatic variable. At run time, each thread's stack contains its own instances of local automatic variables. Different threads may execute the same thread routine. However, they will have different instances of the automatic variables.
3. *Local static variables.* A variable declared in a function with the `static` keyword is a local static variable. A static variable is similar to a global variable. There is exactly one instance, which is shared by all threads.

§13.4.3 Shared Variables

A variable is *shared* if one of its instances can be referenced by multiple threads.

For example, `cnt` is shared by both peer threads while `myid` is not shared. Each peer thread has its own `myid`.

§13.5 Synchronizing Threads with Semaphores

```
/** Figure 13.16 badcnt.c: An improperly synchronized counter program
 *    code/conc/badcnt.c
 */
```

```
#include "csapp.h"
```

```
#define NITERS 100000000 void *count(void *arg);
    /* Thread routine prototype */
```

```
/* shared counter variable */ unsigned int cnt = 0;
```

```
int main() {
    pthread_t tid1, tid2;

    Pthread_create(&tid1, NULL, count, NULL);
```

```

Pthread_create(&tid2, NULL, count, NULL);
Pthread_join(tid1, NULL);
Pthread_join(tid2, NULL);

if (cnt != (unsigned)NITERS*2)
    printf("BOOM! cnt=%d\n", cnt);
else
    printf("OK cnt=%d\n", cnt);
exit(0);
}

/* thread routine */ void *count(void *arg) {
    int i;
    for (i = 0; i < NITERS; i++)
        cnt++;
    return NULL;
}

```

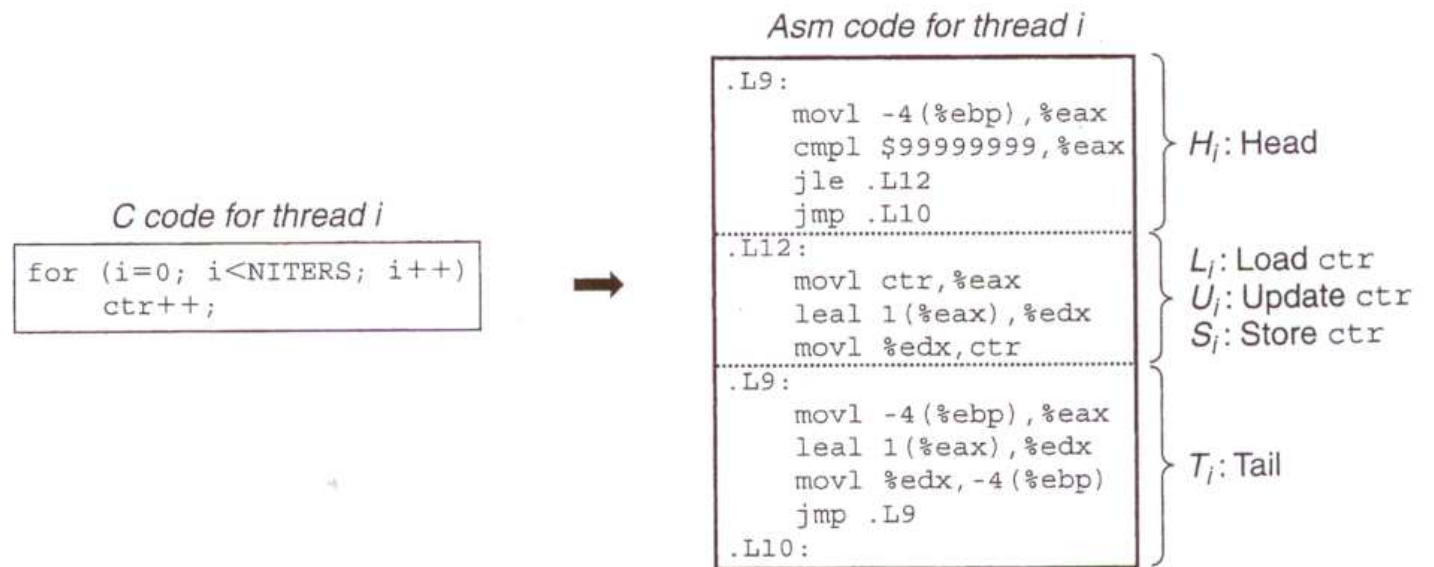
}

When multiple threads attempts to access a shared variable at about the same time, the accesses must be carefully coordinated (i.e., synchronized). Consider the `badcnt.c` program in Figure 13.16. The `cnt` variable is shared by two threads. Each thread will increment `cnt` `NITERS` times. The final value of `cnt` is expected to be $2 \times \text{ITERS}$. However, when we run the program 3 times, we always get the wrong result:

```
unix> ./badcnt
BOOM! ctr=198841183
unix> ./badcnt
BOOM! ctr=198226776
unix> ./badcnt
BOOM! ctr=197655437
```

We will need to examine the assembly code for the `for` loop, which is shown in Figure 13.17.

Figure 13.17
IA32 assembly code for
the counter loop in
`badcnt.c`.



The synchronization problem is due to the fact that increment is not done in a single, atomic step. Rather it is done in several steps. In Figure 13.17, it is done in 3 assembly instructions:

```
.L12
movl    ctr, %eax
```

```
leal    1 (%eax), %edx
movl    %edx, ctr
```

Without proper coordination, it is imaginable that several threads, attempting to increment the same shared variable, may create undesirable final result. Figure 13.18(a) shows an execution order that produces a correct result while Figure 13.18(b) shows a wrong execution order. We can use the *progress graph* for instruction ordering.

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	-	-	0
2	1	L_1	0	-	0
3	1	U_1	1	-	0
4	1	S_1	1	-	1
5	2	H_2	-	-	1
6	2	L_2	-	1	1
7	2	U_2	-	2	1
8	2	S_2	-	2	2
9	2	T_2	-	2	2
10	1	T_1	1	-	2

(a) Correct ordering

Step	Thread	Instr	%eax ₁	%eax ₂	cnt
1	1	H_1	-	-	0
2	1	L_1	0	-	0
3	1	U_1	1	-	0
4	2	H_2	-	-	0
5	2	L_2	-	0	0
6	1	S_1	1	-	1
7	1	T_1	1	-	1
8	2	U_2	-	1	1
9	2	S_2	-	1	1
10	2	T_2	-	1	1

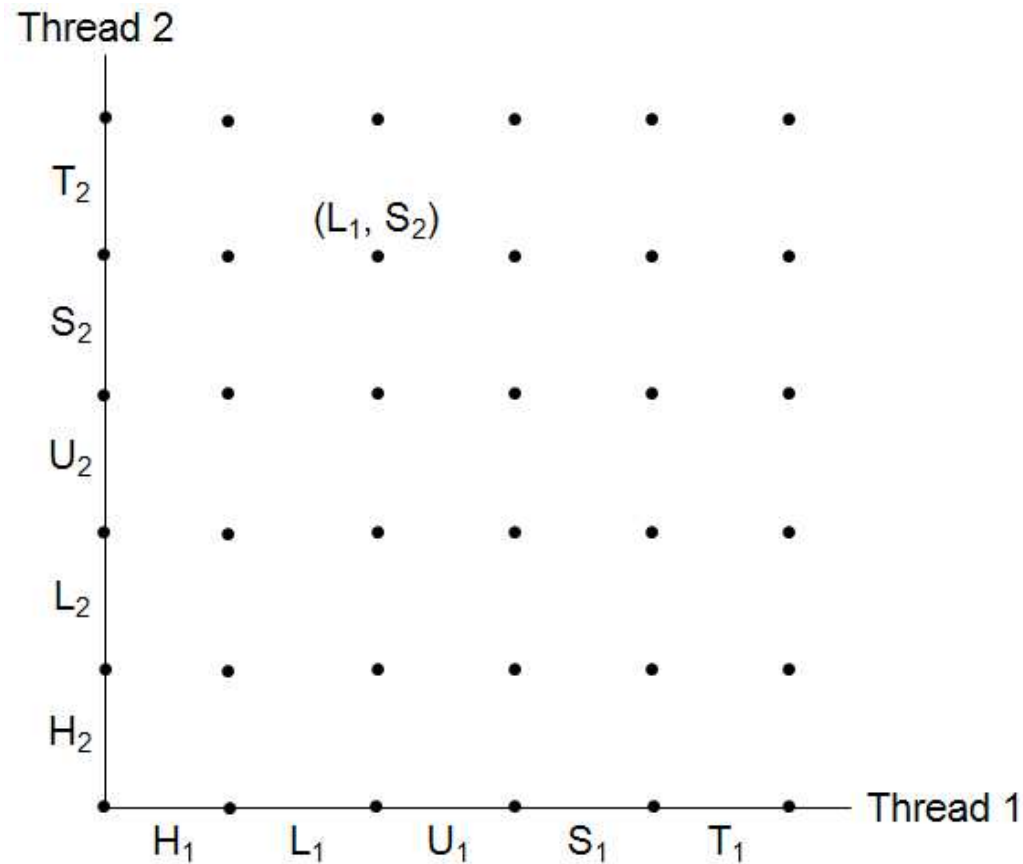
(b) Incorrect ordering

Figure 13.18 Instruction orderings for the first loop iteration in `badcnt.c`.

§13.5.1 Progress Graph

A progress graph models the concurrent execution of n threads with an n -dimensional Cartesian space. Each point $(I_1, I_2, I_3, \dots, I_n)$ represents that state that thread k has completed instruction k . The origin of this graph corresponds to the initial state, in which no thread has completed any instruction.

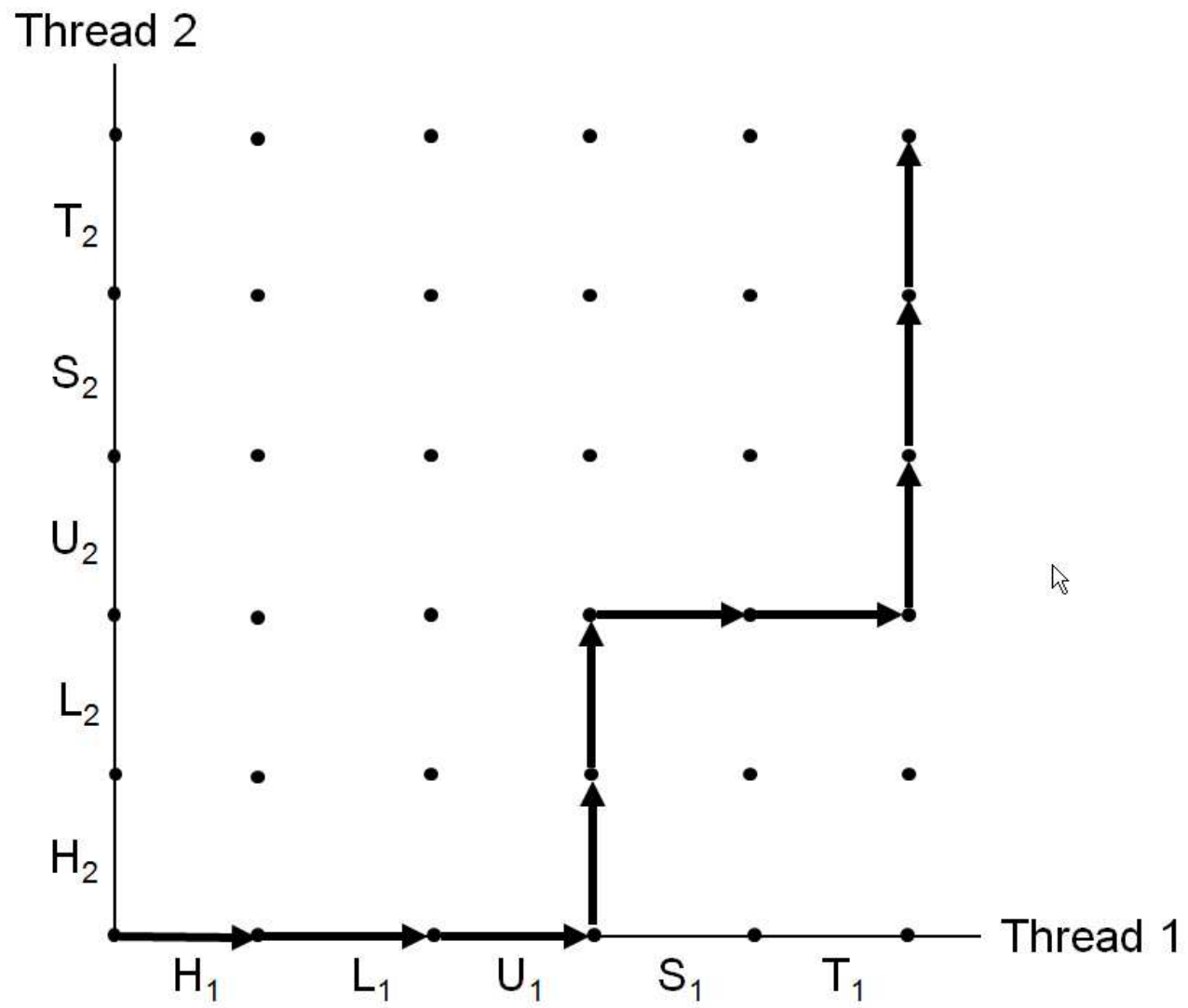
Figure 13.19 shows a 2-dimensional progress graph.



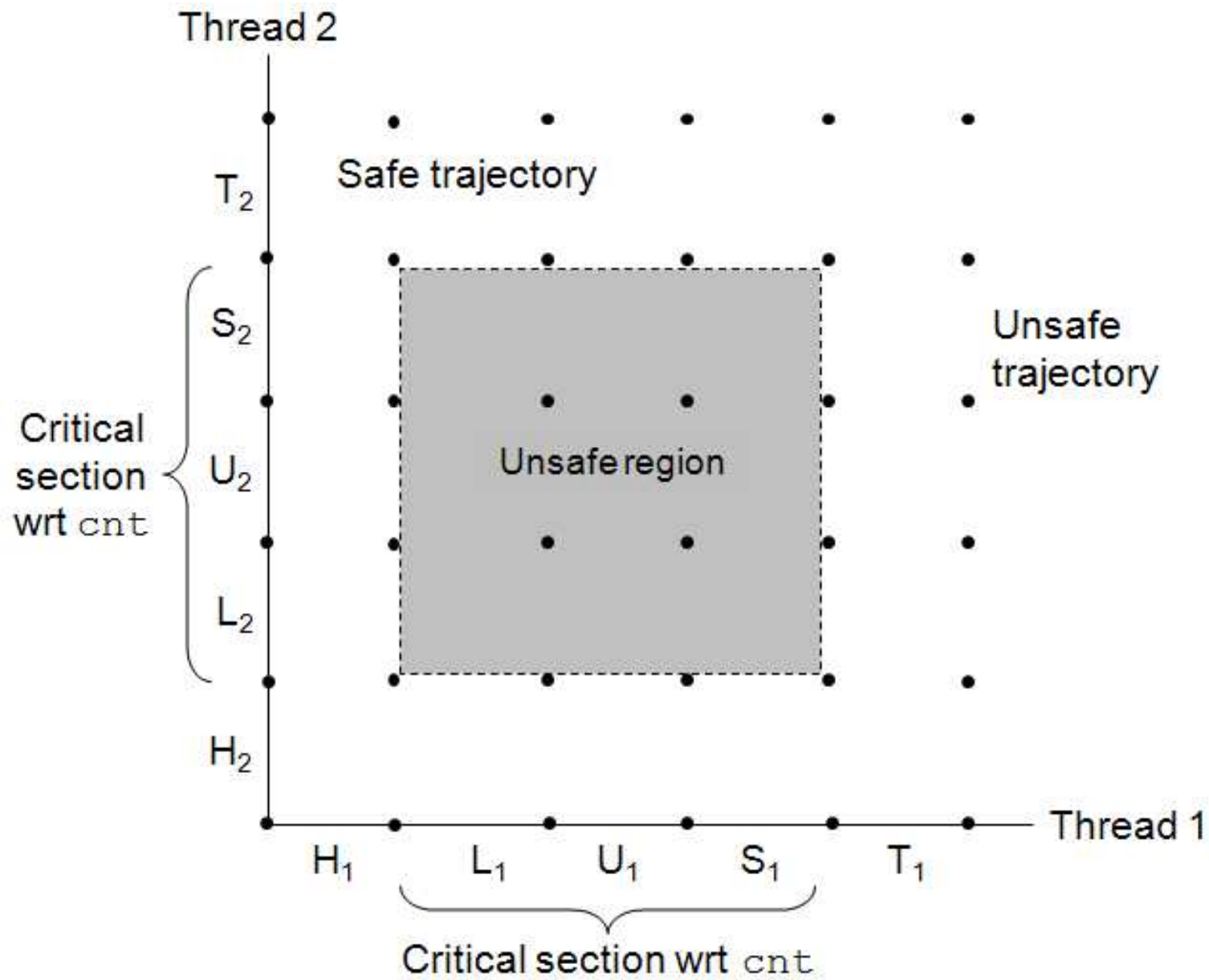
A progress graph models instruction execution as a transition from one state to another. Legal transitions move up or to the right. Two instructions cannot complete at the same time. The execution history is modelled as a *trajectory* through the state space. Figure

13.20 is an trajectory. ^a

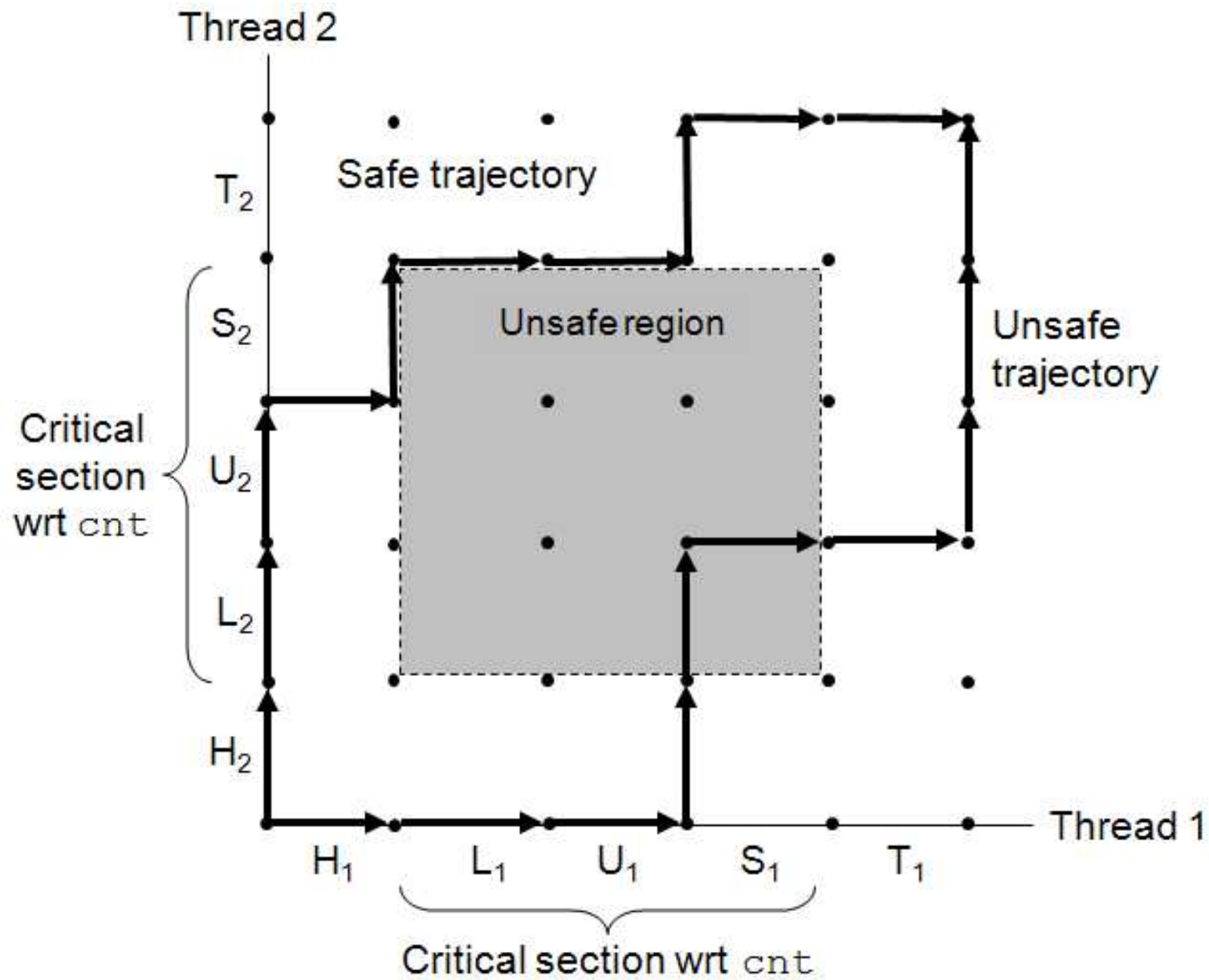
^aIn the above progress graph, the granularity is one instruction. In the more general case, two threads may execute their instruction at (almost) the same time. There is no restriction that one instruction must be completed before the next can start. There are transient states during an instruction execution and other thread may read these transient states.



In order to properly coordinate the access to shared variables, every thread puts its code that manipulate the shared variable in *critical sections*. A critical section should not be interleaved with other critical sections (for the same shared variable) in other threads. The intersection of two critical sections is called an *unsafe region*, which means undesirable result might be produced if control ever enter into this region. Figure 13.21 shows an unsafe region. Note that the unsafe region does not include the points on the boundary.



A trajectory that does not enter the unsafe region is called a *safe trajectory*; otherwise it is an *unsafe trajectory*. Figure 13.22 shows an example of each. We have to synchronize concurrent threads in order to avoid unsafe trajectories, or equivalently, to guarantee safe trajectories.



§13.5.2 Using Semaphores to Access Shared Variables

A semaphore is a special global variable with two operations:

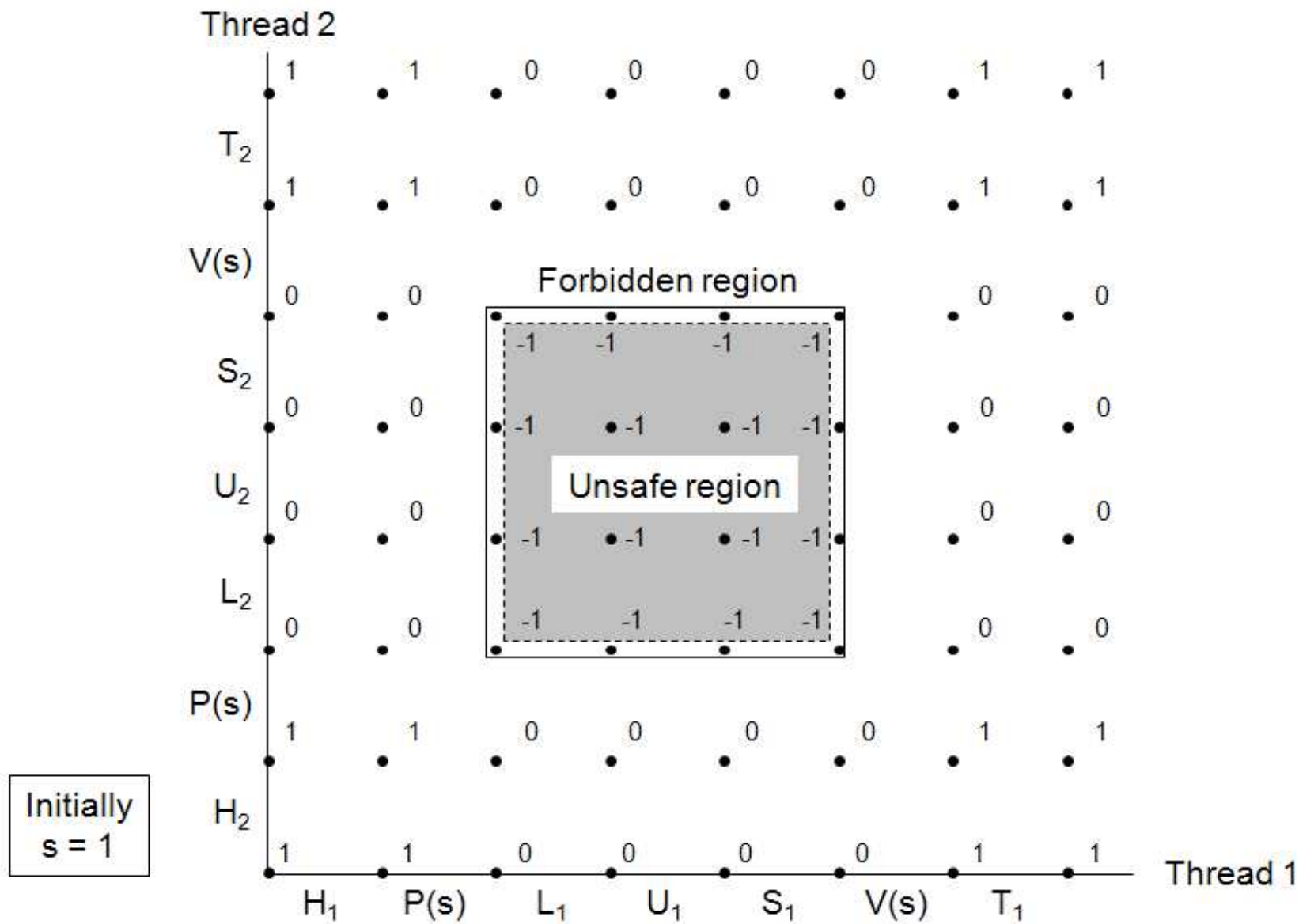
1. $P(s)$: If $s \neq 0$, then $s := s - 1$ and returns immediately. The calling thread continues execution. If $s = 0$, then suspend the thread until s becomes nonzero and then the thread is waken up by a $V(s)$ operation. After restarting, the P operation decrements s and returns control to the caller.
2. $V(s)$: $V(s)$ increments s by 1. If there are any threads blocked by a $P(s)$ operation (waiting for s to become nonzero), $V(s)$ restarts exactly one of them, which then completes its P operation by decrementing s .

The test and decrement operations in P occur atomically, that is, without interruption. The increment operation in V also occurs atomically.

Note that a properly initialized semaphore can never become

negative. We can use P and V to control the trajectories of concurrent programs to avoid unsafe regions. We use a semaphore with initial value 1 for a shared variable and surround every critical section with $P(s)$ and $V(s)$. This kind of semaphore is called a *binary semaphore*.

Figure 13.23 is a progress graph with control by a binary semaphore. Each state (i.e., point) is labelled with the value of the semaphore. The collection of states (points) with a negative semaphore is called the **forbidden region**. Due to the semaphore invariant property, no feasible trajectories (controlled by the semaphore) can ever touch the forbidden region. Because the forbidden region includes all the unsafe regions, no feasible trajectories can ever touch any part of the unsafe regions. Thus, every feasible trajectory is safe. That is, the concurrent thread program correctly increments the counter.



A binary semaphore ensures that at most one thread can execute the code in a critical region at any time. It guarantees mutually exclusive access to the critical sections. For this reason, a semaphore is also called *mutex*, P is called *locking* and V is called *unlocking*.

§13.5.3 Posix Semaphores

Posix standard includes several functions on semaphores. The three basic operations are `sem_init`, `sem_wait` (i.e., P), and `sem_post` (i.e., V).

```
#include <semaphore.h>
int sem_init(sem_t *sem, 0, unsigned int value);
int sem_wait(sem_t *sem);    // P(s)
int sem_post(sem_t *sem);    // V(s)
    // Returns 0 if OK, -1 on error.
```

A program initializes a semaphore with `sem_init`. `value` is the initial value. Then we can use the following wrapper functions for P and V:

```
#include "csapp.h"
void P(sem_t *sem);    // wrapper for sem_wait
```

```
void V(sem_t *sem);    // wrapper for sem_post
```

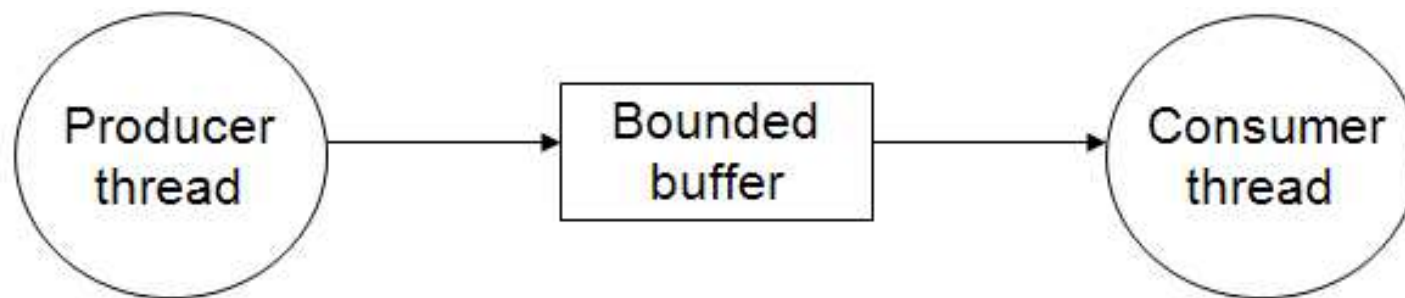
Our program fragment looks like the following segment:

```
sem_t mutex;  
sem_init(&mutex, 0, 1);  
P(&mutex);  
cnt ++;  
V(&mutex);
```

§13.5.4 Using Semaphores to Schedule Shared Resources

A thread can use a semaphore to notify another thread that some condition in the program state has occurred.

A classic example is the *producer-consumer* model,, shown in Figure 13.24. They share a buffer, which stores the produced items that will be consumed. The producer keeps on producing new items while the consumer keeps on consuming them.



To insert or remove items from the buffer, the threads need to update shared variables. In addition to guarantee mutual exclusion, we also need to schedule accesses to the buffer. In this example, the producer can access the shared buffer only if there are empty slots and the consumer can access the shared buffer only if the buffer is not empty.

The producer-consumer paradigm is common. For example, a producer can produce video frames and a consumer can decode the frames and render them on the screen. The shared buffer is a reservoir of slots to the producer and a reservoir of encoded frames to the consumer. This reduces jitters due to differences in individual frames.

Another example is the graphic user interface. The producer keeps on catching mouse and keyboard actions (events) and insert them into a buffer while the consumer takes events from the buffer in some priority-based manner and repaints the screen.

In this section, we will build a simple buffer package `SBUF`, which manipulates buffers of type `sbuf_t`, shown in Figure 13.25. Items are stored in `buf` with `n` slots. `front` and `rear` keep track of the first and last items. Three semaphores control accesses to the buffer: `mutex` provides mutual exclusion. `slots` and `items` semaphores count the numbers of empty slots and available items, respectively.

```
typedef struct {
    int *buf;           /* Buffer array */
    int n;              /* Maximum number of slots */
    int front;         /* buf[(front+1)%n] is first item */
    int rear;          /* buf[rear%n] is last item */
    sem_t mutex;       /* Protects accesses to buf */
    sem_t slots;        /* Counts available slots */
    sem_t items;       /* Counts available items */
} sbuf_t;
```

Figure 13.25 sbuf_t: A sharedbuffer for producer-consumer program

The `sbuf_init` function (Figure 13.26) allocates heap memory for the buffer, sets `front` and `rear` pointers to indicate an empty buffer, and initializes the three semaphores.

```
void sbuf_init(sbuf_t *sp, int n) {
    sp->buf = Calloc(n, sizeof(int));
    sp->n = n;          /* Buffer holds max of n items */
    sp->front = sp->rear = 0; /* Empty buffer iff front == rear */
    Sem_init(&sp->mutex, 0, 1); /* Binary semaphore for locking */
    Sem_init(&sp->slots, 0, n); /* Initially, buf has n empty slots */
    Sem_init(&sp->items, 0, 0); /* Initially, buf has zero data items */
}
```

Figure 13.26 sbuf_int: Initialize a shared buffer

The `sbuf_deinit` function (not shown) frees the buffer storage. The `sbuf_insert` function (Figure 13.27) waits for an available slot, locks the mutex, adds the item, unlocks the mutex, and announces the new item.

```
void sbuf_insert(sbuf_t *sp, int item) {
    P(&sp->slots);           /* Wait for available slot */
    P(&sp->mutex);           /* Lock the buffer */
    sp->buf[(++sp->rear)%(sp->n)] = item; /* Insert the item */
    V(&sp->mutex);           /* Unlock the buffer */
    V(&sp->items);           /* Announce available item */
}
```

Figure 13.27 `sbuf_insert`: Insert an item at the rear of a shared buffer. This function waits until a slot becomes available.

The `sbuf_remove` function (figure 13.28) removes an item from the buffer, if available.

```
int sbuf_remove(sbuf_t *sp) {
    int item;
    P(&sp->items);           /* Wait for available item */
    P(&sp->mutex);           /* Lock the buffer */
    item = sp->buf[(++sp->front)%(sp->n)]; /* Remove the item */
    V(&sp->mutex);           /* Unlock the buffer */
    V(&sp->slots);           /* Announce available slot */
    return item;
}
```

Figure 13.28 `sbuf_remove`: Remove an item from the front of a shared buffer. This function waits until an item becomes available.

Java provides *monitors* for mutual exclusion and scheduling accesses to shared variable. Monitors can be implemented with semaphores.

Pthread Interface defines mutexes and condition variables. Mutexes are for mutual exclusion. Condition variables are for scheduling accesses to shared variables.

§13.6 A Concurrent Server Based on Prethreading

In a concurrent server, a new thread is created for each new request. With a technique called *prethreading*, there is a master thread and several worker threads. The master accepts incoming requests and put them in a buffer. The workers picks up the request from the buffer and services them. After a request is serviced, the worker goes on to the next request. This way, the overhead of creating new threads is reduced.

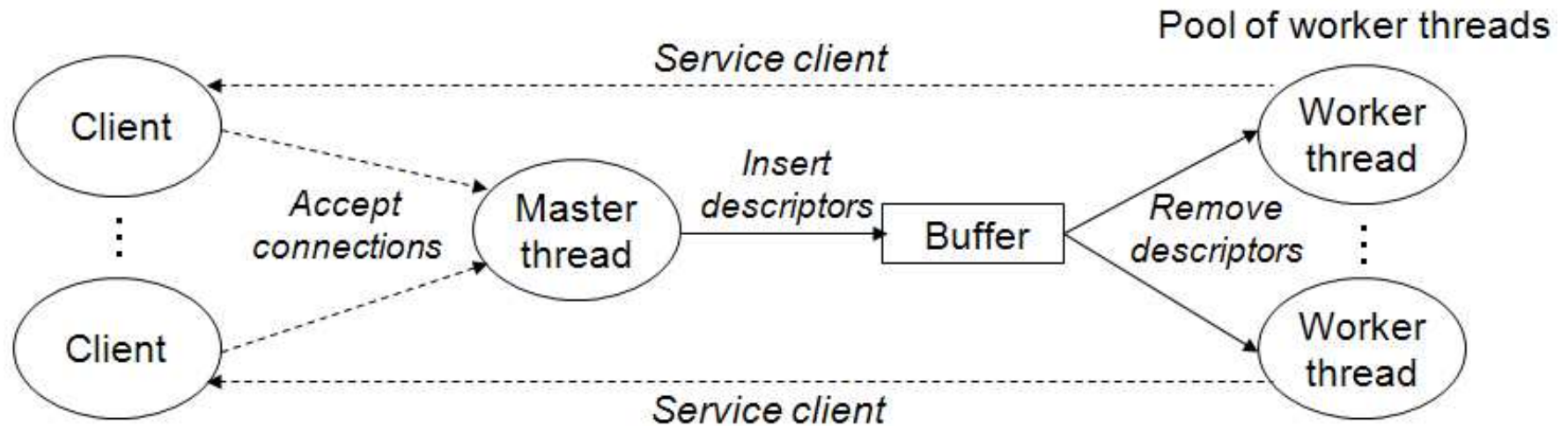


Figure 15: Organization of a pthread concurrent server. A set of existing threads repeatedly remove and process connected descriptors from a shared buffer.

The shared buffer contains connected descriptors. Figure 13.30 shows the code for the main and the worker threads, which makes use of the `sbuf` package. The main thread initializes the `sbuf` package (including the `byte_cnt` counter and the semaphores) and creates some worker threads. It enters a loop that waits for new connection requests and put them in the shared buffer.

```
#include "csapp.h"
#include "sbuf.h"
#define NTHREADS 4
#define SBUFSIZE 16

void echo_cnt(int connfd); void *thread(void *vargp);

sbuf_t sbuf; /* shared buffer of connected descriptors */

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

```

int i, listenfd, connfd, port;
socklen_t clientlen=sizeof(struct sockaddr_in);
struct sockaddr_in clientaddr;
pthread_t tid;

if (argc != 2) {
fprintf(stderr, "usage: %s <port>\n", argv[0]);
exit(0);
}

port = atoi(argv[1]);
sbuf_init(&sbuf, SBUFSIZE); //line:conc:pre:initsbuf
listenfd = Open_listenfd(port);

for (i = 0; i < NTHREADS; i++) /* Create worker threads */
    Pthread_create(&tid, NULL, thread, NULL);

```

```

while (1) {
    connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    sbuf_insert(&sbuf, connfd); /* Insert connfd in buffer */
}

}

void *thread(void *vargp) {
    Pthread_detach(pthread_self());
    while (1) {
        int connfd = sbuf_remove(&sbuf); /* Remove connfd from buffer */
        echo_cnt(connfd);                /* Service client */
        Close(connfd);
    }
}
}

```

Figure 13.30 A prethreaded concurrent echo server. The server

uses a producer-consumer model with one producer and multiple consumers.

The `echo_cnt` function (figure 13.31) records the total number of bytes received from all clients in the global variable `byte_cnt`.

The `byte_cnt` counter can be initialized in two ways:

1. the main thread calls the initialization function during the start-up stage. All programs that makes use of the `sbuf` package must *remember* to do this.
2. The initialization function is called via the `pthread_once` function, which is again called from the `echo_cnt` function. The disadvantage is the overhead. Everytime the `echo_cnt` function is invoked, the `pthread_once` function will be executed though only the first time is useful. Later invocations do nothing useful.

```
#include "csapp.h"

static int byte_cnt; /* byte counter */
static sem_t mutex; /*and the mutex that protects it */

static void init_echo_cnt(void) {
    Sem_init(&mutex, 0, 1);
    byte_cnt = 0;
}

void echo_cnt(int connfd) {
    int n;
    char buf[MAXLINE];
    rio_t rio;
    static pthread_once_t once = PTHREAD_ONCE_INIT;
```

```

Pthread_once(&once, init_echo_cnt);
Rio_readinitb(&rio, connfd);
while((n = Rio_readlineb(&rio, buf, MAXLINE)) != 0) {
    P(&mutex);
    byte_cnt += n; //line:conc:pre:cntaccess1
    printf("thread %d received %d (%d total) bytes on fd %d\n",
          (int) pthread_self(), n, byte_cnt, connfd);
    V(&mutex);
    Rio_writen(connfd, buf, n);
}
}

```

Figure 13.31 echo_cnt: A version of echo that counts all bytes received from clients.

§13.7 Other Concurrency Issues

§13.7.1 Thread Safety

A function is *thread-safe* if and only if it always produces correct results when called repeatedly and concurrently from multiple concurrent threads. Otherwise it is *thread-unsafe*.

How to guarantee a function is thread-safe?

There are four classes of thread-unsafe functions:

- Class 1: Functions that do not protect shared variables. A shared variable that is accessed arbitrarily by concurrent threads will have unpredictable value. A remedy is to protect each shared variable with a semaphore. This change is easy. A disadvantage is that synchronization may slow down the function.
- Class 2: Functions that keep state across multiple invocation.

For example, a pseudorandom number generator has a state which changes every time the generator runs. The `rand` function in Figure 13.32 is not thread-safe because the result of current invocation depends on an intermediate result from the previous iteration. When `rand` is invoked by concurrent threads without proper control, two threads may even obtain the same random number.

The only way to make such a function thread-safe is to rewrite it so that it does not make use of any `static` data. The state information is passed through the parameters. This might be a problem in a large program since there may be many call sites.

```
unsigned int next = 1;

/* rand - return pseudo-random integer on 0..32767 */
int rand(void) {
    next = next*1103515245 + 12345;
```

```
    return (unsigned int)(next/65536) % 32768;
}

/* srand - set seed for rand() */
void srand(unsigned int seed) {
    next = seed;
}
```

Figure 13.32 A thread-unsafe pseudorandom number generator [40].

- Class 3: Functions that return a pointer to a **static** variable. Some functions compute a result in a **static** structure and return a pointer to it. These functions could become disasters because the **static** structure is shared (silently) by several threads.

One way to remedy this problem is to have the caller to

allocate a new structure and pass it to the intended functions. Therefore, concurrent threads will *not* share the data structure. If a thread-unsafe function is difficult or impossible to modify, we can use a different *lock-and-copy* approach. A mutex will be created for the thread-unsafe function. At each call site, first lock the mutex, call the function, allocate new storage, and copy the result to the new storage, and then unlock the mutex. Furthermore, we may create a thread-safe wrapper for the original thread-unsafe function. The wrapper does exactly the above operations. Figure 13.33 is an example.

```
struct hostent *gethostbyname_ts(char *hostname) {
    struct hostent *sharedp, *unsharedp;

    unsharedp = Malloc(sizeof(struct hostent));
    P(&mutex);
    sharedp = gethostbyname(hostname);
```

```
    *unsharedp = *sharedp; /* copy shared struct to private struct
return unsharedp;
}
```

Figure 13.33 `gethostbyname_ts`: A thread-safe wrapper for `gethostbyname`. Use the lock-and-copy techniques to call a class 2 thread-unsafe function.

- Class 4: Functions that call thread-unsafe functions. If a function f calls a thread-unsafe function g , is f thread-safe? If g belongs to class 2 (it depends on states across multiple invocations), then f is also unsafe. However, if g belongs to class 1 or class 3, f can be thread-safe if the call site and the resulting data shared structure are protected with a mutex. In Figure 13.33, we use lock-and-copy to write a thread-safe function that calls a thread-unsafe function.

§13.7.2 Reentrancy

Reentrant functions are those that do not reference any shared data when they are called by multiple threads. Reentrant functions are thread-safe. Their relation is shown in Figure 13.34 below.



Reentrant functions are usually more efficient than non-reentrant thread-safe functions because they require no synchronization. Furthermore, the only way to convert a class-2 unsafe function into

a thread-safe one is to the reentrant `rand_r` function modified from Figure 13.32. The key change is the static `next` variable is replaced with a pointer that is passed in by the caller.

If all function arguments are passed by value (i.e., no pointers) and all data references are to local automatic stack variables (i.e., no references to static or global variables), then the function is *explicitly reentrant*, in the sense that it is reentrant regardless of how it is called.

On the other hand, if some parameters are passed by reference (i.e., pointers), then the function is *implicitly reentrant* in the sense that it is only reentrant if the calling threads are careful to pass pointers to non-shared data. The `rand_r` function in Figure 13.35 is implicitly reentrant. That is, reentrancy is a property of both the caller and the callee.

The `gethostbyname_ts` function in Figure 13.33 is thread-safe but not reentrant.

```
int rand_r(unsigned int *nextp) {  
    *nextp = *nextp * 1103515245 + 12345;  
    return (unsigned int)(*nextp / 65536) % 32768;  
}
```

Figure 13.35 rand_r: A reentrant version of the rand function from Figure 13.32.

§13.7.3 Using Existing Library Functions in Threaded Programs

Most Unix functions and the functions defined in the standard C library (such as `malloc`, `free`, `realloc`, `printf`, `scanf`) are thread-safe with only a few exceptions, shown in Figure 13.36.

Thread-unsafe function	Thread-unsafe class	Unix thread-safe version
<code>rand</code>	2	<code>rand_r</code>
<code>strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>

thread-unsafe func.	thread-unsafe class	thread-safe func.
<code>rand</code>	2	<code>rand_r</code>
<code>strtok</code>	2	<code>strtok_r</code>
<code>asctime</code>	3	<code>asctime_r</code>
<code>ctime</code>	3	<code>ctime_r</code>
<code>gethostbyaddr</code>	3	<code>gethostbyaddr_r</code>
<code>gethostbyname</code>	3	<code>gethostbyname_r</code>
<code>inet_ntoa</code>	3	(none)
<code>localtime</code>	3	<code>localtime_r</code>

Except `rand` and `strtok`, all other thread-unsafe functions in Figure 13.36 belongs to class 3 that returns a pointer to a static variable.

When we need to call these functions, we can use the lock-and-copy

method, which suffers the overhead of synchronization.

Unix also provides reentrant versions of most thread-unsafe functions.

§13.7.4 Races

A *race* is a condition that thread 1 must reach point x before thread 2 reaches point y in the control flow graph. Otherwise the result would be incorrect. Races usually occur because programmers assume that the threads will take some particular trajectory through the execution state space, forgetting that threaded programs must work correctly for any feasible trajectory.

```
#include "csapp.h"
#define N 4

void *thread(void *vargp);

int main() {
    pthread_t tid[N];
    int i;
```

```

    for (i = 0; i < N; i++)
        Pthread_create(&tid[i], NULL, thread, &i);
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}

```

Figure 13.37 A program with a race.

Consider the program in Figure 13.37. The main thread creates four peer threads, passing each a pointer to a unique integer ID. Each peer thread copies the ID to a local variable and prints a message. The output of the program is wrong, shown below:

```
unix> ./race
Hello from thread 1
Hello from thread 3
Hello from thread 2
Hello from thread 3
```

The problem is due to a race between the main thread (line 12) and a peer thread (line 21). The main thread passes an integer pointer (`&i`) and the peer thread dereferences it (`*((int *)vargp)`). The peer thread should dereference the pointer before the main thread pass the integer pointer (`&i`) to the next peer thread. Whether the output is correct also depends on how the OS schedules the

threads. Thus sometimes, we might get the correct answers.

To eliminate the race, the main thread can dynamically allocate a separate block for each integer ID, and passes the peer thread a pointer to this block. The revised code is in Figure 13.38. Note that the thread routine must free the block in order to avoid memory leak.

```
#include "csapp.h"
#define N 4

void *thread(void *vargp);

int main() {
    pthread_t tid[N];
    int i, *ptr;

    for (i = 0; i < N; i++) {
        ptr = Malloc(sizeof(int));
        *ptr = i;
        Pthread_create(&tid[i], NULL, thread, ptr);
    }
    for (i = 0; i < N; i++)
        Pthread_join(tid[i], NULL);
}
```

```
    exit(0);
}

/* thread routine */
void *thread(void *vargp) {
    int myid = *((int *)vargp);
    Free(vargp);
    printf("Hello from thread %d\n", myid);
    return NULL;
}
```

Figure 13.38 A correct version of the program in Figure 13.37 without a race.

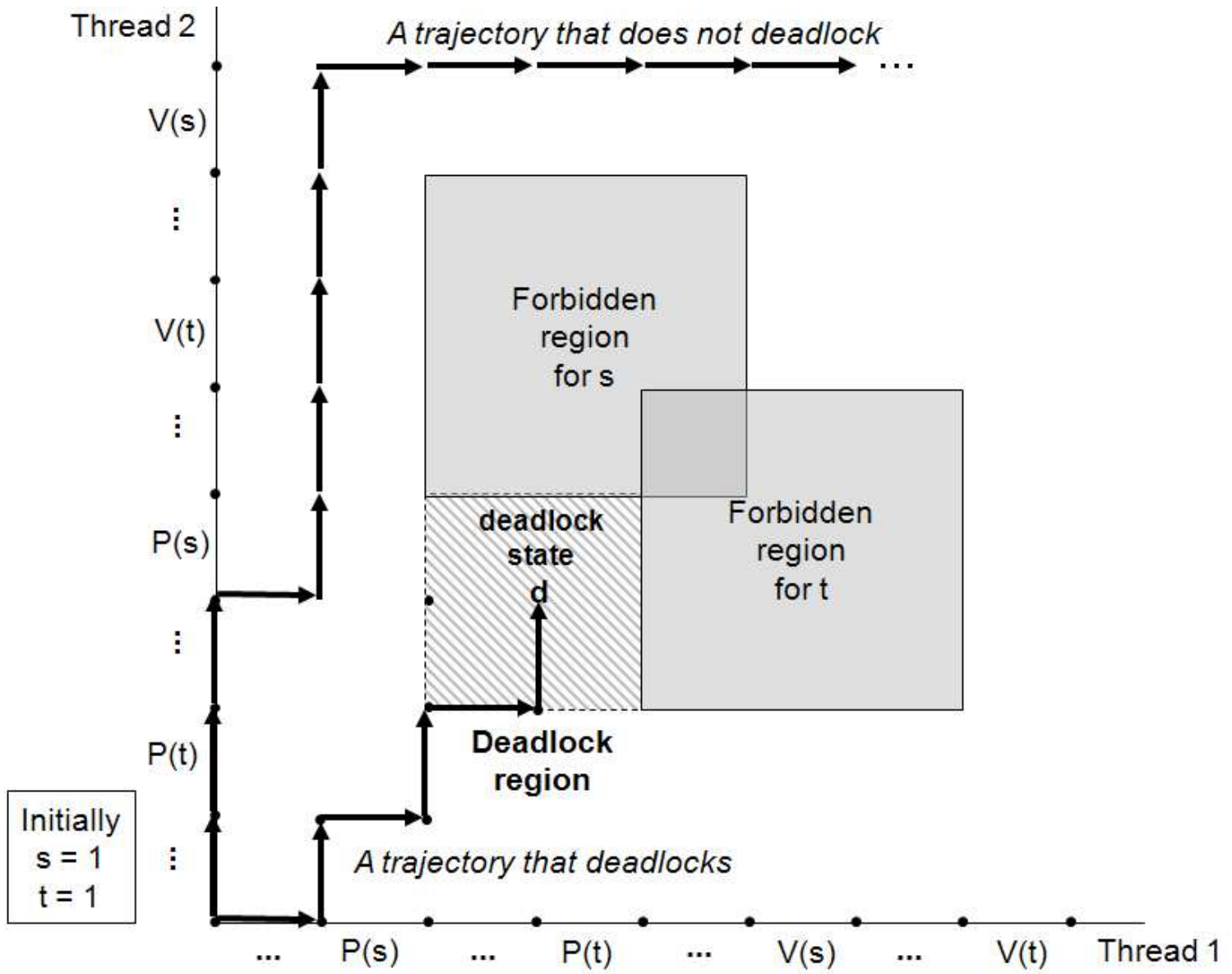
§13.7.5 Deadlocks

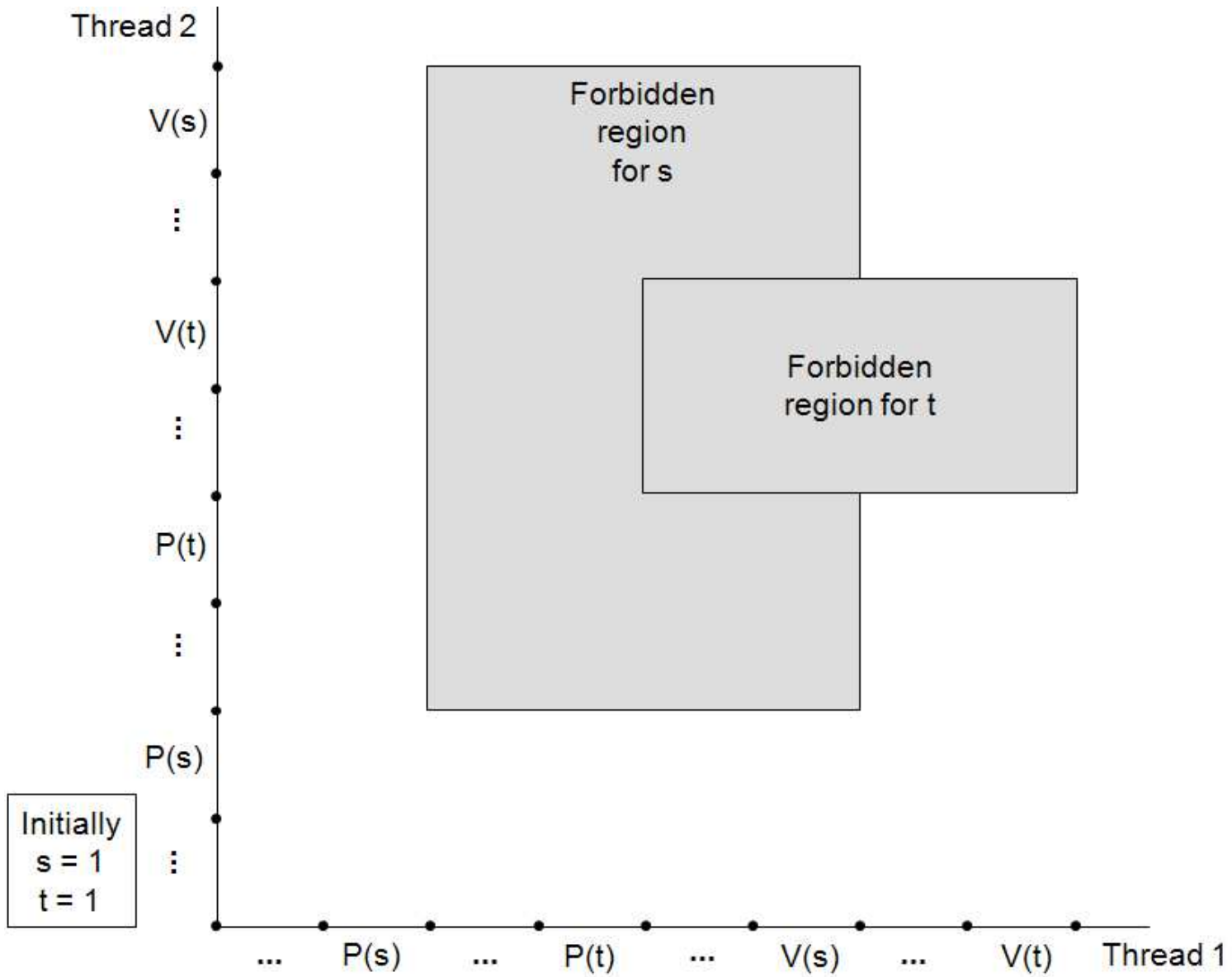
Semaphores may create *deadlocks*. Consider the progress graph in Figure 13.39. There are two semaphores s and t . Thread 1 does $P(s)$ and then $P(t)$. Thread 2 does $P(t)$ and then $P(s)$. Once the trajectory enters the deadlock state d , no further progress is possible because the (overlapping) forbidden regions block progress in every legal direction.

The overlapping forbidden regions induce the deadlock region.

Though there is a deadlock region, deadlocks may not always happen. Furthermore, deadlocks may not be repeatable. This causes difficulty in debugging and testing threaded programs.

A simple rule to avoid deadlocks is to order the semaphores as s_1, s_2, \dots . When a thread needs several semaphores, it must acquire its semaphores in this order. For example, in Figure 13.40, every thread always locks s first then locks t .





§13.8 Summary

A concurrent program consists of a collection of logical flows that overlap in time. We studied three approaches:

1. processes
2. I/O multiplexing
3. threads

Processes are scheduled automatically by the kernel. They require explicit IPC mechanisms in order to share data.

Event-driven programs use I/O multiplexing to explicitly schedule the logical flows. But a program runs in a single process and can share data easily.

Threads are a hybrid of the above methods.

There are mechanisms for synchronizing concurrent threads/processes when they access shared variables.

Functions must be *thread-safe* in a concurrent program otherwise errors might occur.

Reentrant functions are thread-safe and they do not need synchronization.

Races and deadlocks are common problems in concurrent programming.

Index

Posix Threads, 70