Network Programming

BITS Pilani
Pilani Campus

K Hari Babu
Department of Computer Science & Information Systems

innovate achieve lead

BITS Pilani
Pilani Campus

Outline

Outline

- Threads
  - Overview
  - Creation
  - Termination
  - Join
  - Detach
- Thread synchronization
  - Mutexes
  - Condition variables
- Server Design
- Multi Process Concurrency
  - Preforking models
  - Prethreading models
- Single Process Concurrency
  - Signal driven I/O
  - epoll()
- Concurrency in UDP

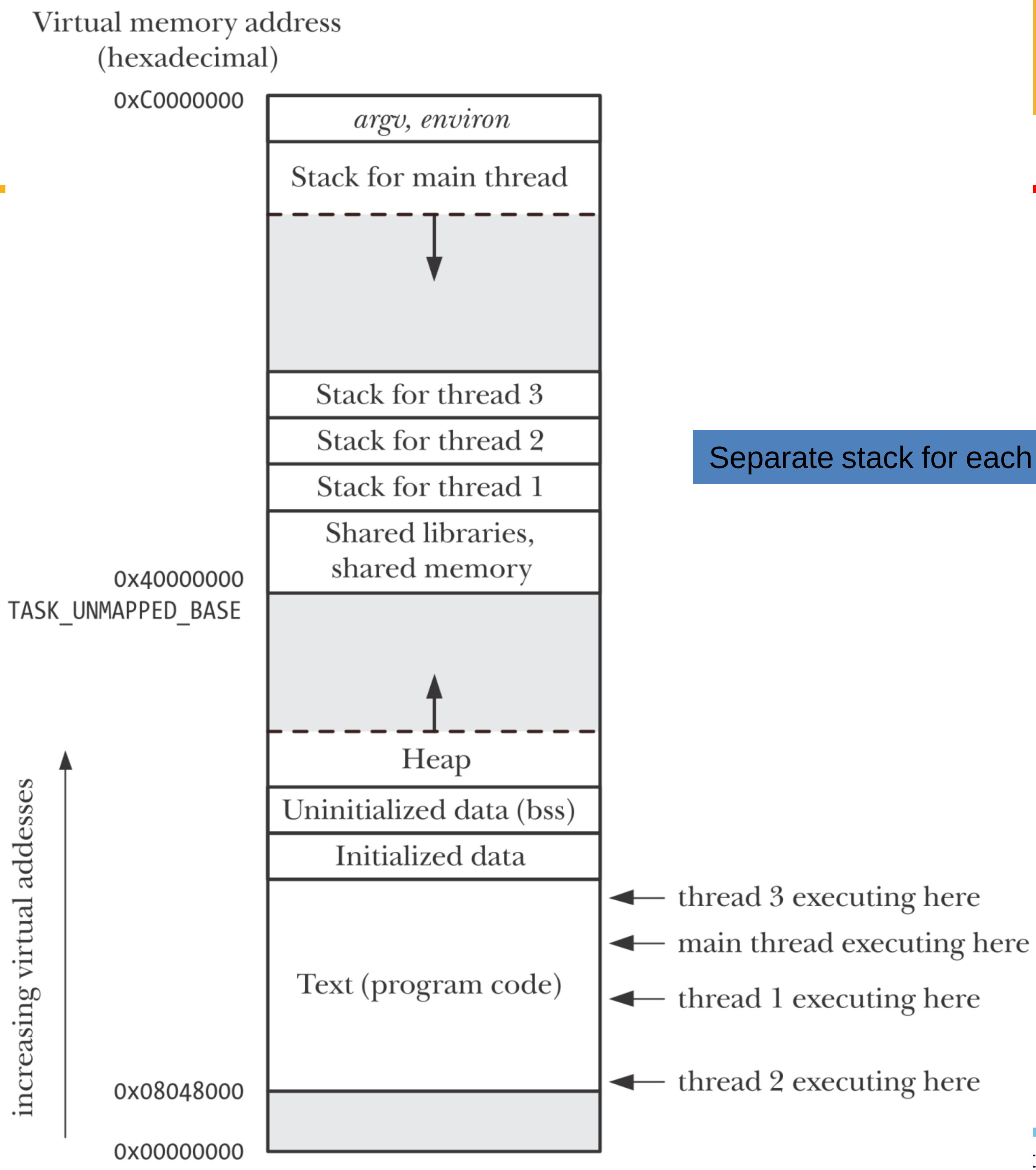
BITS Pilani
Pilani Campus

Pthreads

T1: ch 26

Threads

- A thread is a line of execution.
- A single process can contain multiple threads.
  - The threads in a process can execute concurrently. On a multiprocessor system, multiple threads can execute parallel.
- Problems with multiple processes in an application
  - Exchange of information among processes requires IPC mechanism.
    - All threads in a process share process's data segments (all global variables) , and heap apart from text segment. Stack is individual to each thread.
    - But synchronization mechanisms are required to control access.
  - Process creation with fork() is relatively expensive (even with copy-on-write).
    - Thread creation is 10 times faster. No page-table duplication. No memory duplication.



Separate stack for each thread

Multiple threads run concurrently

Threads

- Threads share
  - heap, data, text segments, global variables, file descriptors, CWD, user, group ids, signal handlers, signal dispositions
- Individual
  - stack, registers, program counters, *errno*, signal mask, priority
- *errno* is local to each thread.
  - Normally sys calls return -1 on error and set errno. But Pthread API calls return 0 on success and >0 on failure. They do not set errno.
    - errno is set when a sys call is directly called within a thread.
- Compiling pthread applications.

```
gcc  pthread.c -lpthread
```

  - The program is linked with the libpthread library.

Pthreads API

- Thread Management
    - creation, detach, join, exit
    - POSIX requires all threads are created as joinable threads
- Thread Synchronization
    - join
    - mutex (i.e. semapore with value of 1)
    - semaphore
    - condition variables

Thread Creation

- When a program is started by exec, a single thread is created, called the initial thread or main thread. Additional threads are created by *pthread_create*.

```c
#include <pthread.h>
int pthread_create(pthread_t *thread, const pthread_attr_t * attr,
                   void *(* start )(void *), void * arg );
//Returns 0 on success, or a positive error number on error
```

  - *start:* this is the function, the thread will start executing.
  - *thread*: this is the thread id, filled by kernel.
  - *attr*: normally NULL.
    - Each thread has numerous attributes: its priority, its initial stack size, whether it should be a daemon thread or not etc.
    - When a thread is created, we can specify these attributes by initializing a pthread_attr_t variable that overrides the default.
  - *arg*: argument to the function.

Joining with Terminated Thread

- We can wait for a given thread to terminate by calling pthread_join.
  - pthread_create is similar to fork, and pthread_join is similar to waitpid.

```
include <pthread.h>
int pthread_join(pthread_t  thread , void ** retval );
//Returns 0 on success, or a positive error number on error
```

  - If *retval* is a non NULL pointer, then it receives status of a thread specified by pthread_exit().
  - A thread can call join on any thread in the process. No parent-child relationship in threads.
  - A non-detatched thread must be joined by some thread, otherwise it will lead to zombie thread. Resources will be held up in the kernel.

Detaching a Thread

- By default a thread is joinable. If we do not join, kernel will store the status of the thread.
- If we do not care about the status of the thread, then we can detach the thread.
  - System will automatically cleanup when the thread terminates.

```
#include <pthread.h>
int pthread_detach(pthread_t  thread );
//Returns 0 on success, or a positive error number on error
```

  - Detaching a thread doesn’t make it immune to exit() in another thread or a return in the main thread.
  - pthread_detach() simply controls what happens after a thread terminates, not how or when it terminates.
- A thread can detach itself.

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

More Pthread Functions

- pthread_self Function
  - Each thread has an ID that identifies it within a given process. The thread ID is returned by pthread_create().
  - A thread fetches this value for itself using pthread_self().

```c
#include <pthread.h>
pthread_t pthread_self(void);
//Returns the thread ID of the calling thread
```

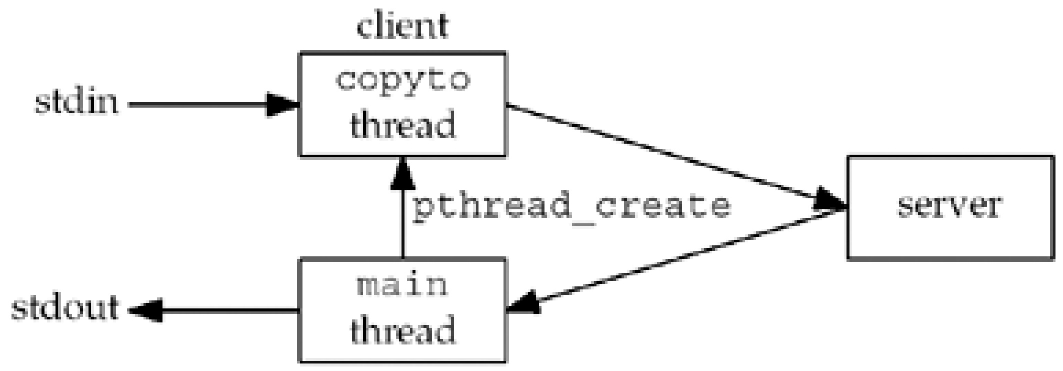
- pthread_exit Function

```c
#include <pthread.h>
void pthread_exit(void * retval );
```

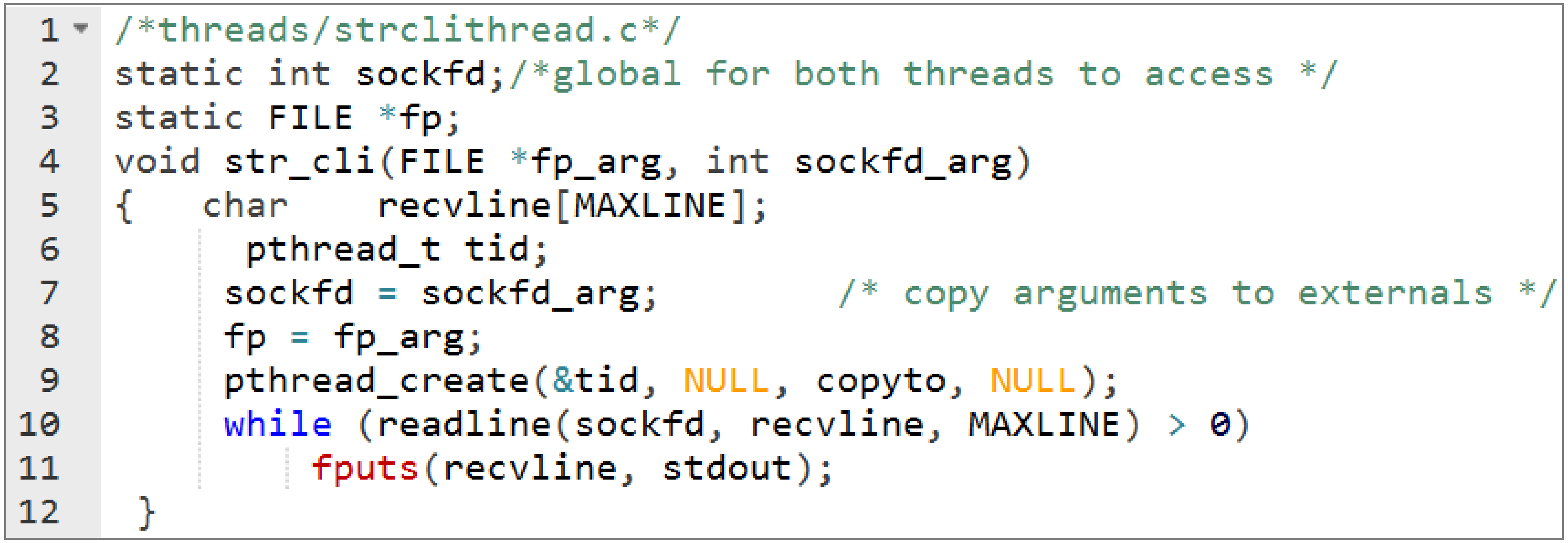
  - The retval argument specifies the return value for the thread.
  - Calling pthread_exit() is equivalent to performing a return in the thread's start function.
    - If the main thread calls pthread_exit() instead of calling exit() or performing a return , then the other threads continue to execute.

Batch Mode Client Using Threads

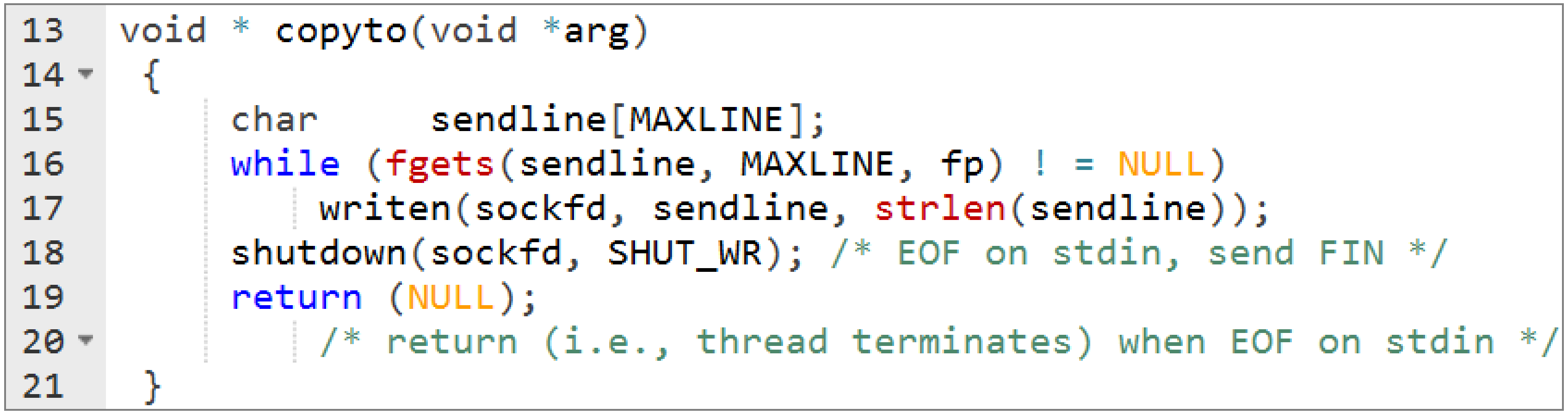
- Client uses two threads
  - *Main* thread: data transfer from socket to *stdout*
  - *copyto* thread: data transfer from *stdin* to socket.



Batch Mode Client Using Threads



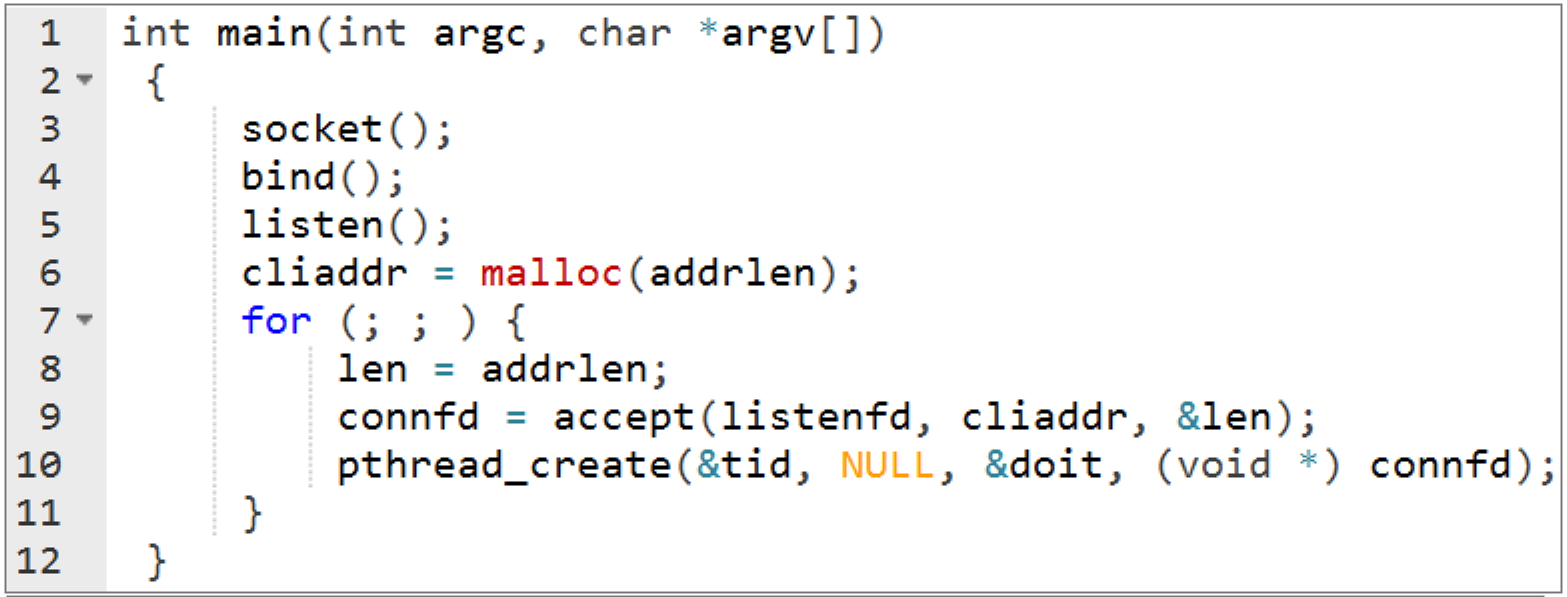
```c
/*threads/strclithread.c*/
static int sockfd;/*global for both threads to access */
static FILE *fp;
void str_cli(FILE *fp_arg, int sockfd_arg)
{   char    recvline[MAXLINE];
     pthread_t tid;
    sockfd = sockfd_arg;          /* copy arguments to externals */
    fp = fp_arg;
    pthread_create(&tid, NULL, copyto, NULL);
    while (readline(sockfd, recvline, MAXLINE) > 0)
        fputs(recvline, stdout);
 }
```



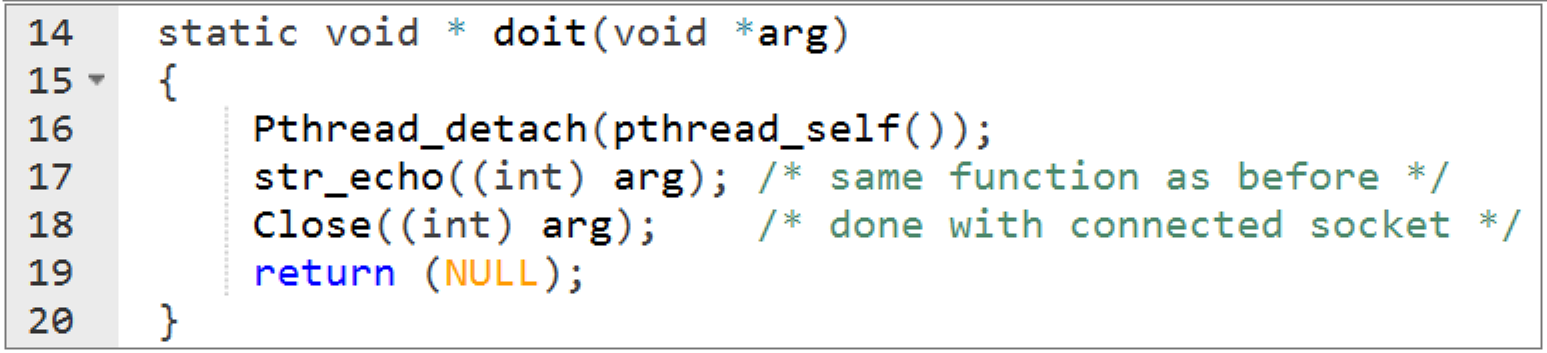
```c
void * copyto(void *arg)
 {
    char     sendline[MAXLINE];
    while (fgets(sendline, MAXLINE, fp) ! = NULL)
        writen(sockfd, sendline, strlen(sendline));
    shutdown(sockfd, SHUT_WR); /* EOF on stdin, send FIN */
    return (NULL);
        /* return (i.e., thread terminates) when EOF on stdin */
 }
```

TCP Echo Server Using Threads

- Server creates a new thread every time it accepts a new connection through accept().



```c
int main(int argc, char *argv[])
{
    socket();
    bind();
    listen();
    cliaddr = malloc(addrlen);
    for (; ; ) {
        len = addrlen;
        connfd = accept(listenfd, cliaddr, &len);
        pthread_create(&tid, NULL, &doit, (void *) connfd);
    }
}
```

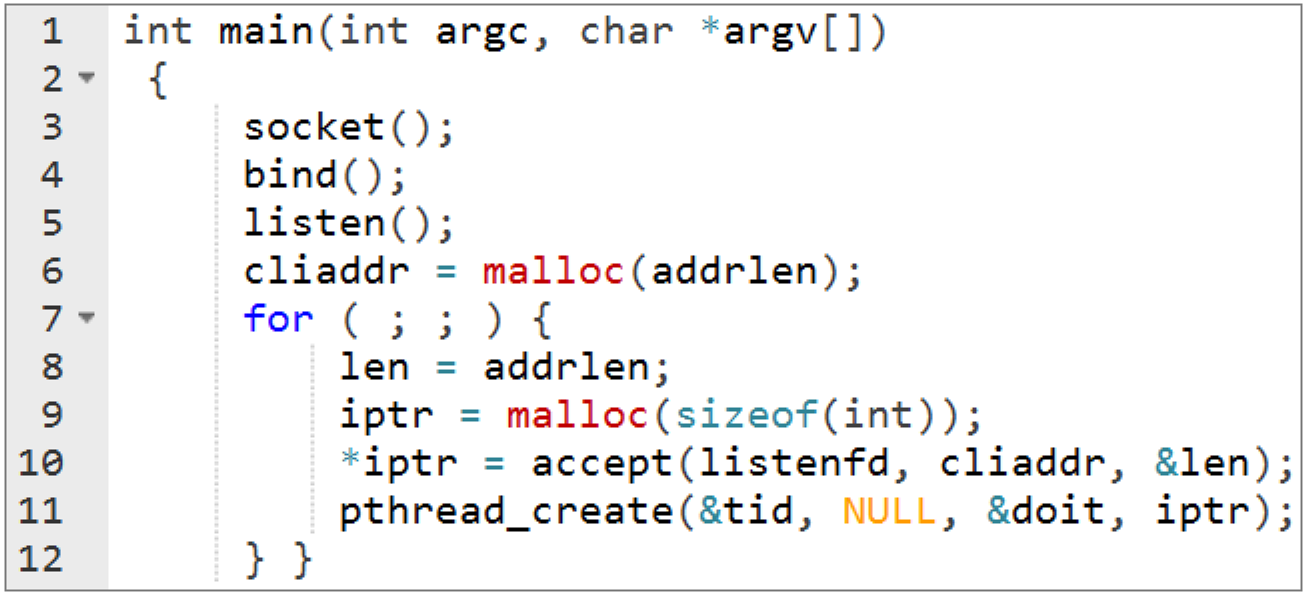


```c
static void * doit(void *arg)
{
    Pthread_detach(pthread_self());
    str_echo((int) arg); /* same function as before */
    Close((int) arg);    /* done with connected socket */
    return (NULL);
}
```

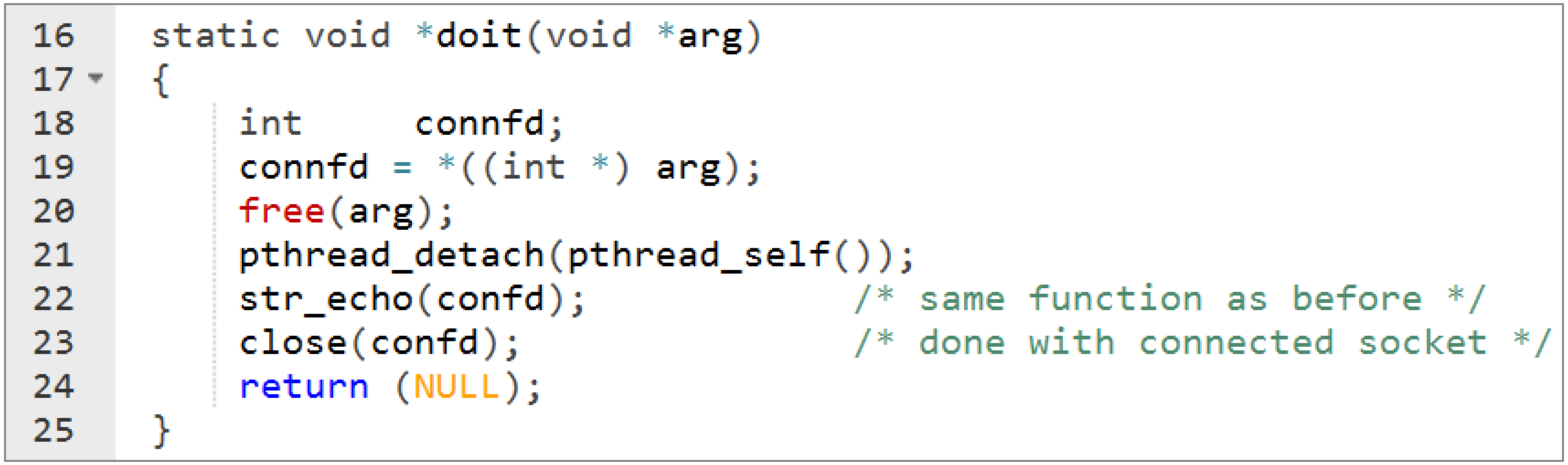
Passing Arguments to New Threads

- Problem with the code in the previous slide:
    - According to ANSI, *connfd* can't be casted to void*. It is not guaranteed to work on all systems.
    - Passing address of *connfd* also not going to work. Because two threads may end up working on the same *connfd* value.
- Each time we call accept, we first call *malloc()* and allocate space for an integer variable, the connected descriptor.
    - This gives each thread its own copy of the connected descriptor.
- The thread fetches the value of the connected descriptor and then calls free to release the memory.

Passing Arguments to New Threads



```c
int main(int argc, char *argv[])
 {
     socket();
     bind();
     listen();
     cliaddr = malloc(addrlen);
     for ( ; ; ) {
         len = addrlen;
         iptr = malloc(sizeof(int));
         *iptr = accept(listenfd, cliaddr, &len);
         pthread_create(&tid, NULL, &doit, iptr);
     } }
```

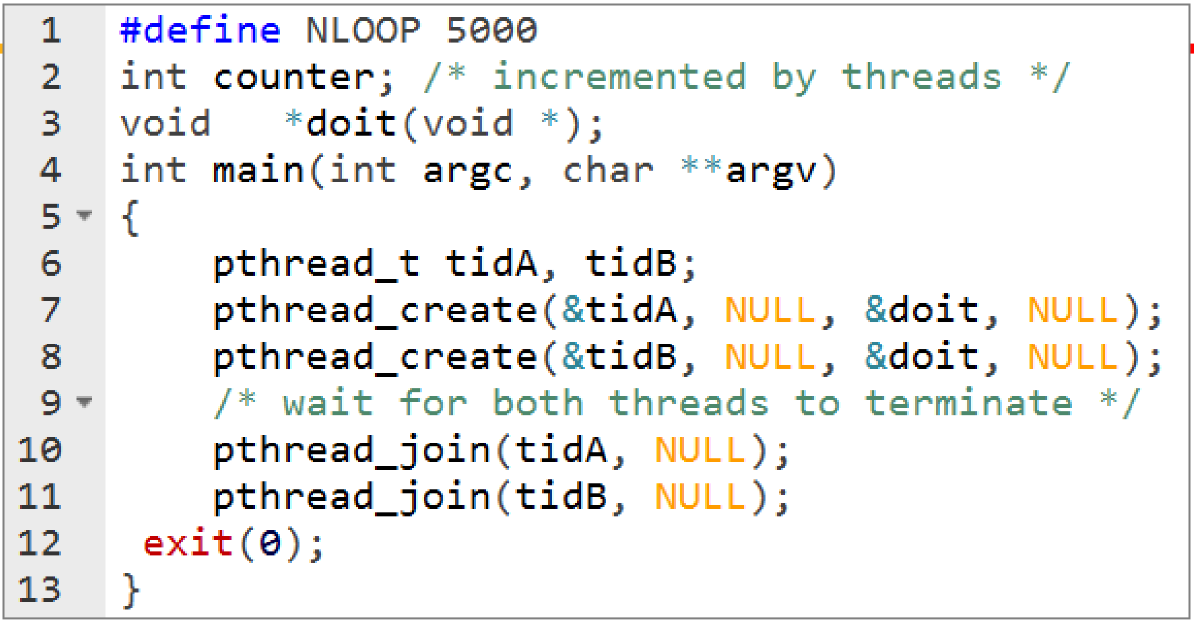


```c
 static void *doit(void *arg)
 {
     int     connfd;
     connfd = *((int *) arg);
     free(arg);
     pthread_detach(pthread_self());
     str_echo(confd);                /* same function as before */
     close(confd);                   /* done with connected socket */
     return (NULL);
 }
```

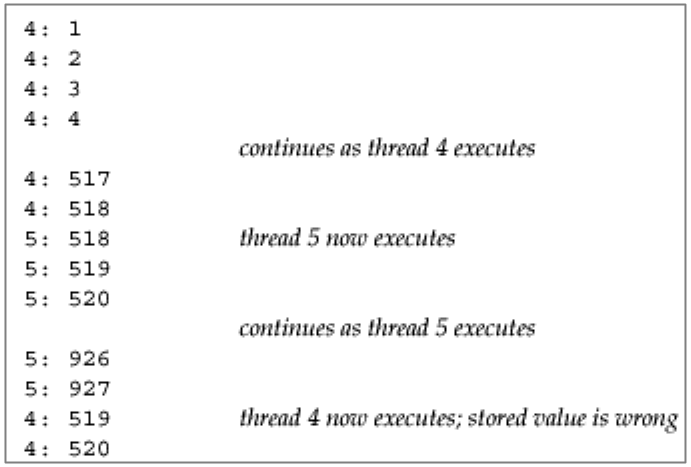
Protecting Accesses to Shared Variables: Mutexes

- The term critical section refers to a section of code that accesses a shared resource and whose execution should be atomic.
- Its execution should not be interrupted by another thread that simultaneously accesses the same shared resource.
- Pthreads provide mutexes for protecting critical section.
  - Each mutex has two states: locked, unlocked.
  - At most one thread may hold lock on a mutex.
  - When a thread locks a mutex, it becomes the owner of that mutex. Only the mutex owner can unlock the mutex.

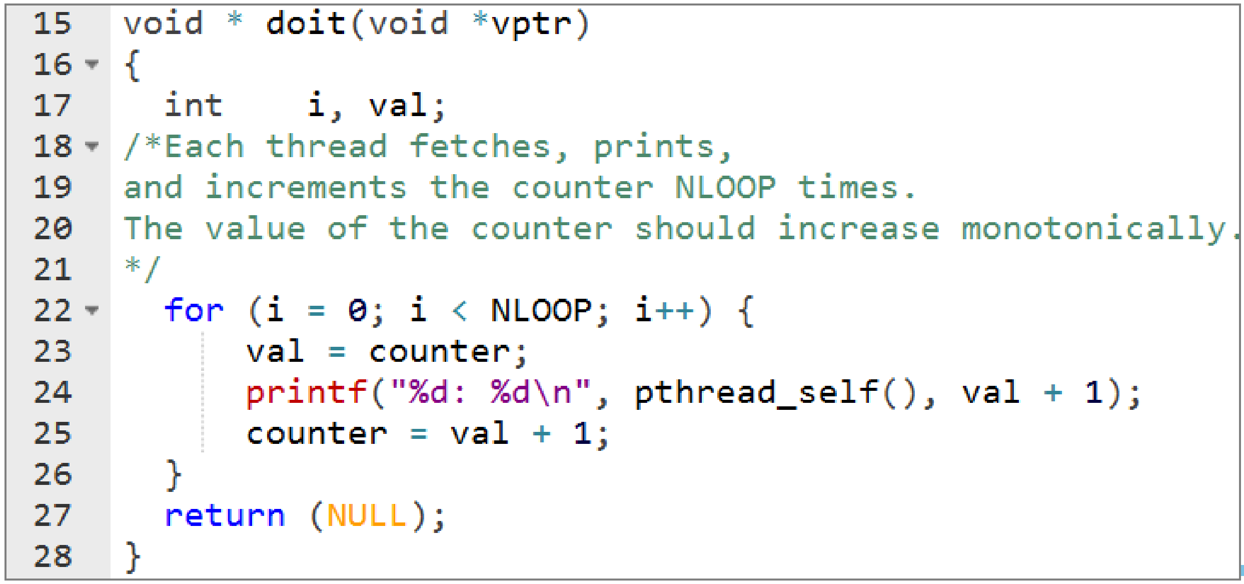
Critical Section Problem



```c
#define NLOOP 5000
int counter; /* incremented by threads */
void   *doit(void *);
int main(int argc, char **argv)
{
    pthread_t tidA, tidB;
    pthread_create(&tidA, NULL, &doit, NULL);
    pthread_create(&tidB, NULL, &doit, NULL);
    /* wait for both threads to terminate */
    pthread_join(tidA, NULL);
    pthread_join(tidB, NULL);
 exit(0);
}
```



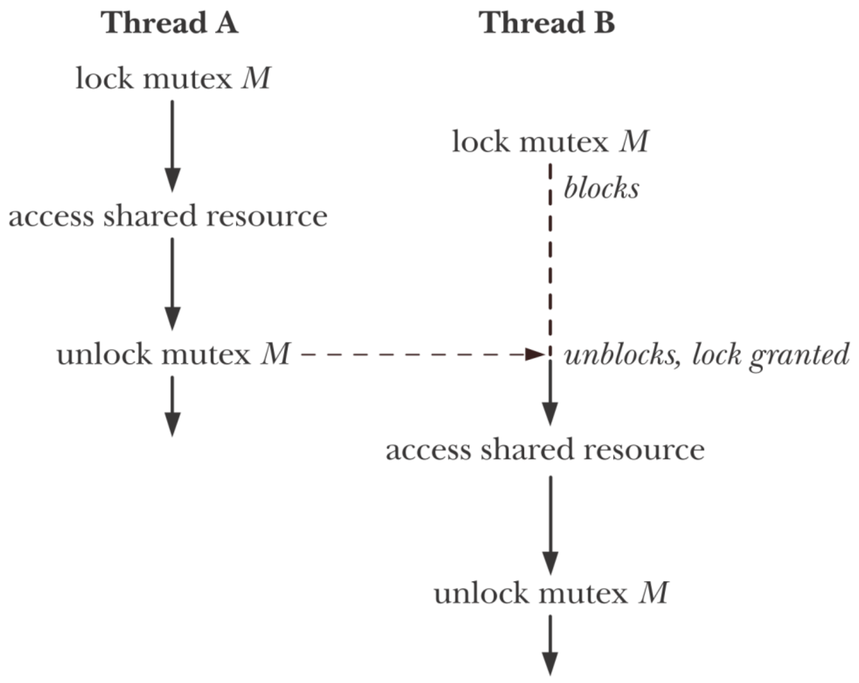
```
4:  1
4:  2
4:  3
4:  4
                    continues as thread 4 executes

4:  517
4:  518
5:  518             thread 5 now executes
5:  519
5:  520
                    continues as thread 5 executes

5:  926
5:  927
4:  519             thread 4 now executes; stored value is wrong
4:  520
```



```c
void * doit(void *vptr)
{
  int    i, val;
/*Each thread fetches, prints,
and increments the counter NLOOP times.
The value of the counter should increase monotonically.
*/
  for (i = 0; i < NLOOP; i++) {
      val = counter;
      printf("%d: %d\n", pthread_self(), val + 1);
      counter = val + 1;
  }
  return (NULL);
}
```

Using Mutex

- Each thread employs the following protocol for accessing a resource:
  - lock the mutex for the shared resource;
  - access the shared resource; and
  - unlock the mutex.



Using Mutex

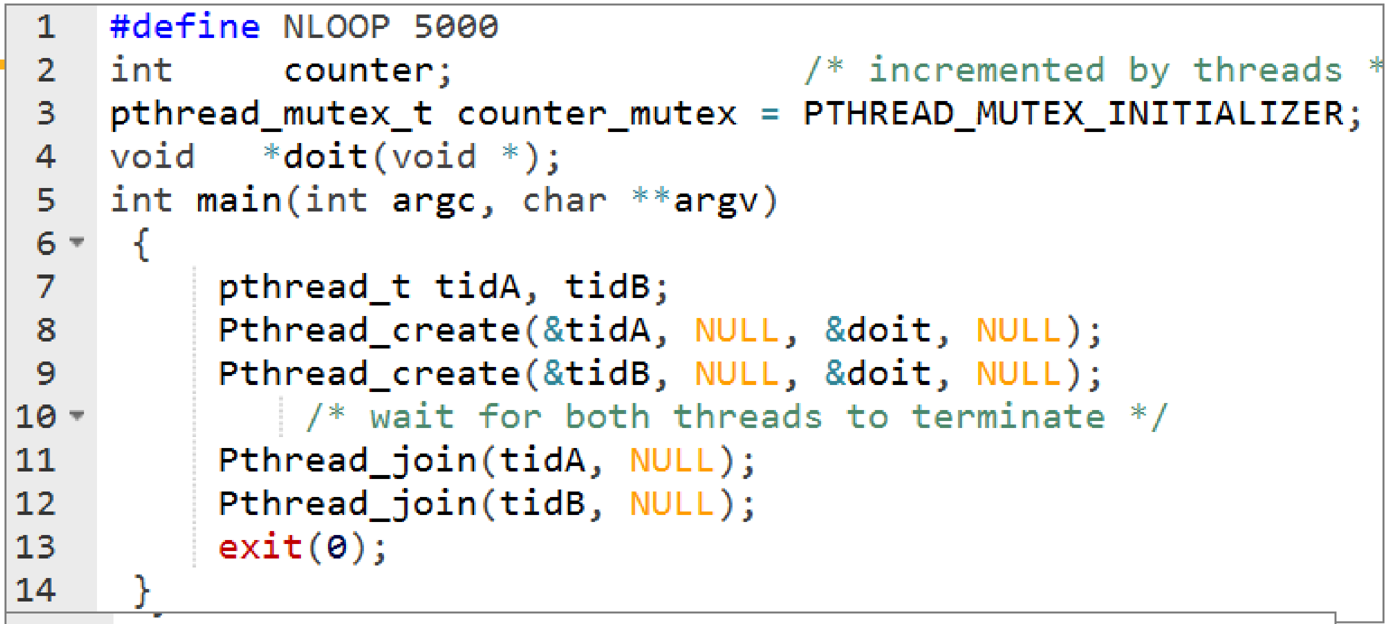
- A mutex is a variable of the type pthread_mutex_t. Mutex must always be initialized.
  - For a statically allocated mutex

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
```

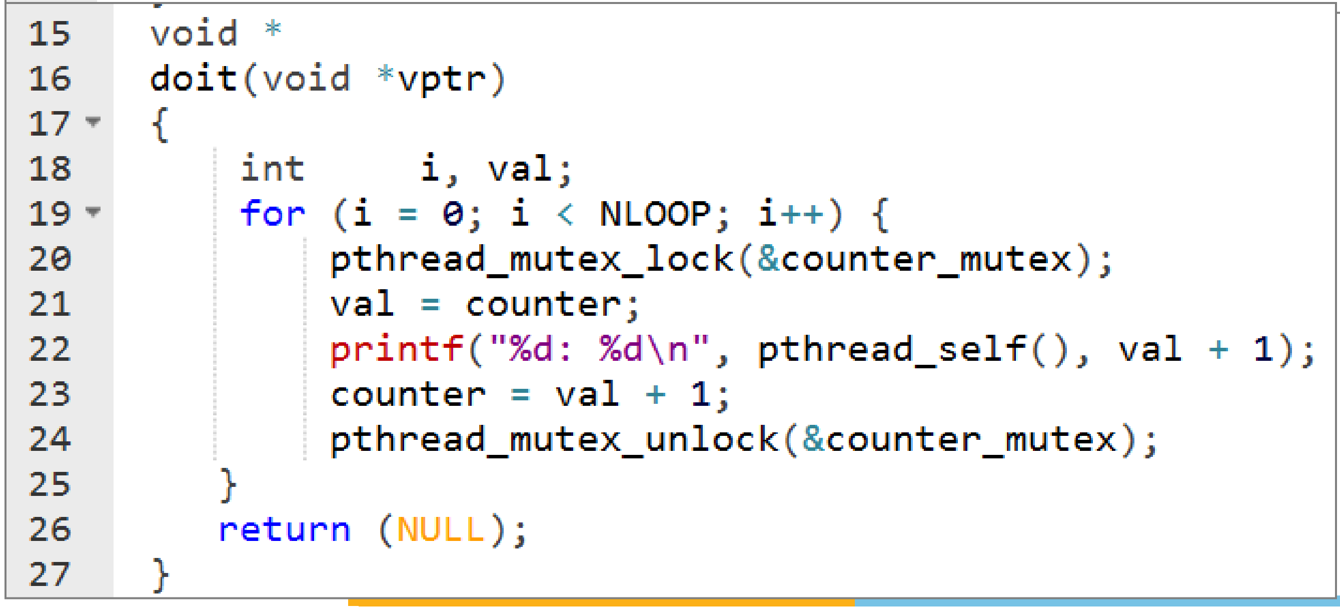
```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t * mutex );
int pthread_mutex_unlock(pthread_mutex_t * mutex );
//Both return 0 on success, or a positive error number on error
```

  - The pthread_mutex_trylock() function is the same as pthread_mutex_lock(), except that if the mutex is currently locked, pthread_mutex_trylock() fails, returning the error EBUSY.

Using Mutexes



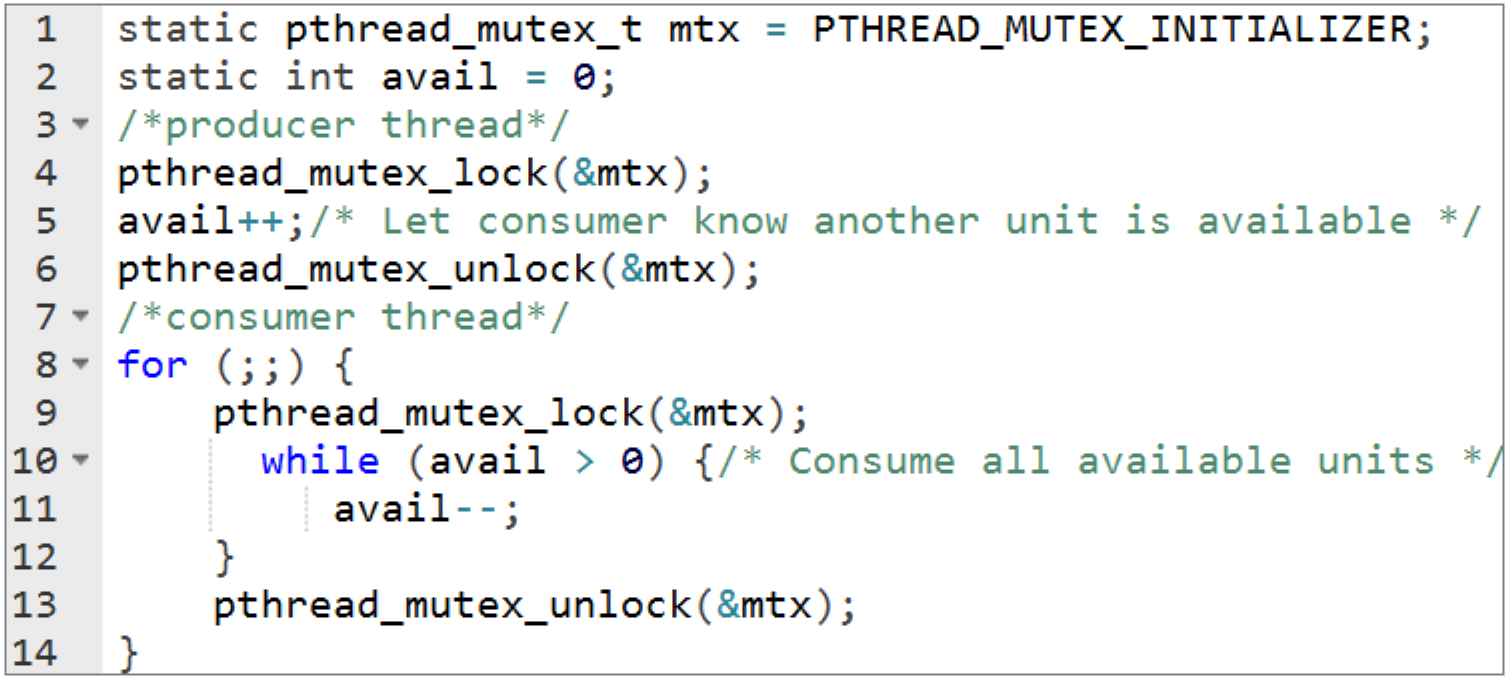
```c
#define NLOOP 5000
int     counter;                /* incremented by threads *
pthread_mutex_t counter_mutex = PTHREAD_MUTEX_INITIALIZER;
void    *doit(void *);
int main(int argc, char **argv)
 {
     pthread_t tidA, tidB;
     Pthread_create(&tidA, NULL, &doit, NULL);
     Pthread_create(&tidB, NULL, &doit, NULL);
         /* wait for both threads to terminate */
     Pthread_join(tidA, NULL);
     Pthread_join(tidB, NULL);
     exit(0);
 }
```



```c
void *
doit(void *vptr)
{
    int     i, val;
    for (i = 0; i < NLOOP; i++) {
        pthread_mutex_lock(&counter_mutex);
        val = counter;
        printf("%d: %d\n", pthread_self(), val + 1);
        counter = val + 1;
        pthread_mutex_unlock(&counter_mutex);
    }
    return (NULL);
}
```

Condition Variables

- A mutex is fine to prevent simultaneous access to a shared variable, but we need something else to let us go to sleep waiting for some condition to occur.



```
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static int avail = 0;
/*producer thread*/
pthread_mutex_lock(&mtx);
avail++;/* Let consumer know another unit is available */
pthread_mutex_unlock(&mtx);
/*consumer thread*/
for (;;) {
    pthread_mutex_lock(&mtx);
      while (avail > 0) {/* Consume all available units */
         avail--;
    }
    pthread_mutex_unlock(&mtx);
}
```

- The above code works, but it wastes CPU time, because the consumer thread continually loops, checking the state of the variable *avail*. A condition variable remedies this problem.

Condition Variables

- Condition variable allows a thread to sleep (wait) until another thread notifies (signals) it that it must do something.
- A condition variable is always used in conjunction with a mutex.
- The mutex provides mutual exclusion for accessing the shared variable, while the condition variable is used to signal changes in the variable’s state.

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

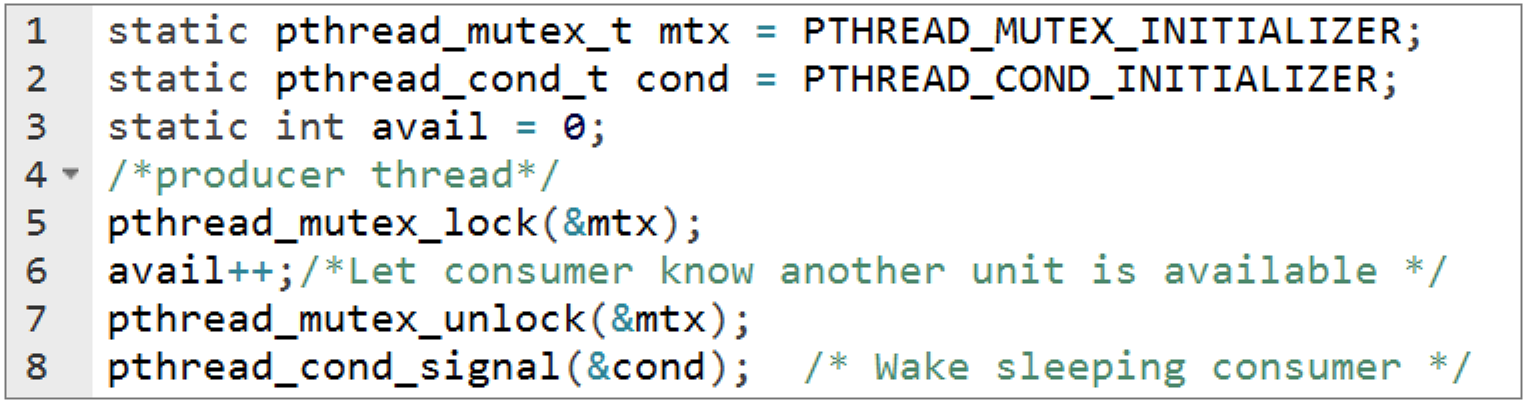
```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t * cond );
int pthread_cond_broadcast(pthread_cond_t * cond );
int pthread_cond_wait(pthread_cond_t * cond , pthread_mutex_t * mutex );
//All return 0 on success, or a positive error number on error
```

  - Broadcast wakes up all blocked threads. Each will go through the code. Used when there is different tasks done for a particular condition.

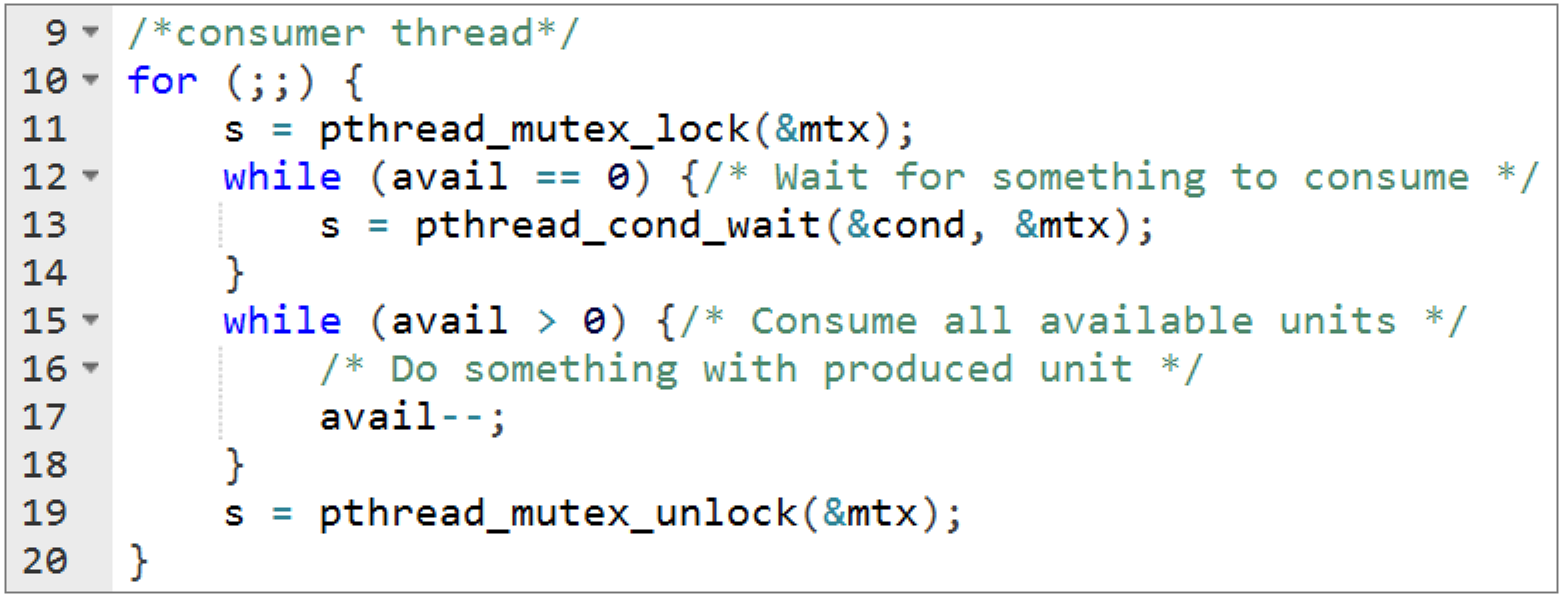
Using Condition Variables

- Why mutex is associated with condition variable?
  - The thread locks the mutex in preparation for checking the state of the shared variable.
  - The state of the shared variable is checked.
  - If the shared variable is not in the desired state, then the thread must unlock the mutex (so that other threads can access the shared variable) before it goes to sleep on the condition variable.
    - Done atomically
  - When the thread is reawakened because the condition variable has been signaled, the mutex must once more be locked, since, typically, the thread then immediately accesses the shared variable.
- it is not possible for some other thread to acquire the mutex and signal the condition variable before the thread calling pthread_cond_wait() has blocked on the condition variable.

Using Condition Variables



```c
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
static int avail = 0;
/*producer thread*/
pthread_mutex_lock(&mtx);
avail++;/*Let consumer know another unit is available */
pthread_mutex_unlock(&mtx);
pthread_cond_signal(&cond);  /* Wake sleeping consumer */
```



```c
/*consumer thread*/
for (;;) {
    s = pthread_mutex_lock(&mtx);
    while (avail == 0) {/* Wait for something to consume */
        s = pthread_cond_wait(&cond, &mtx);
    }
    while (avail > 0) {/* Consume all available units */
        /* Do something with produced unit */
        avail--;
    }
    s = pthread_mutex_unlock(&mtx);
}
```

BITS Pilani
Pilani Campus

Web client: Using threads

Threads : web client

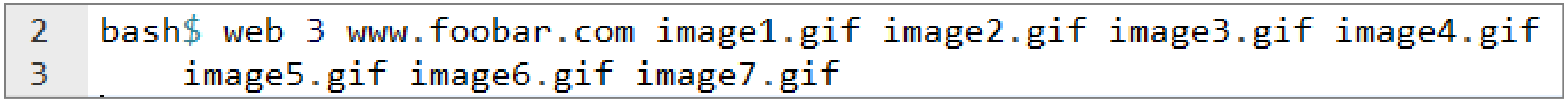
- The client establishes an HTTP connection with a Web server and fetches a home page.
- On that page are often numerous references to other Web pages.
- Instead of fetching these other pages serially, one at a time, the client can fetch more than one at the same time, using multiple connections, one per thread.

Threads : web client

- We have designed a web client handling multiple simultaneous connections using non-blocking connect.
- Now we will design using threads:
  - With threads, we can leave the sockets in their default blocking mode.
  - Create one thread per connection.
  - Each thread can block in its call to connect. Kernel will schedule threads that are ready.

Threads : web client

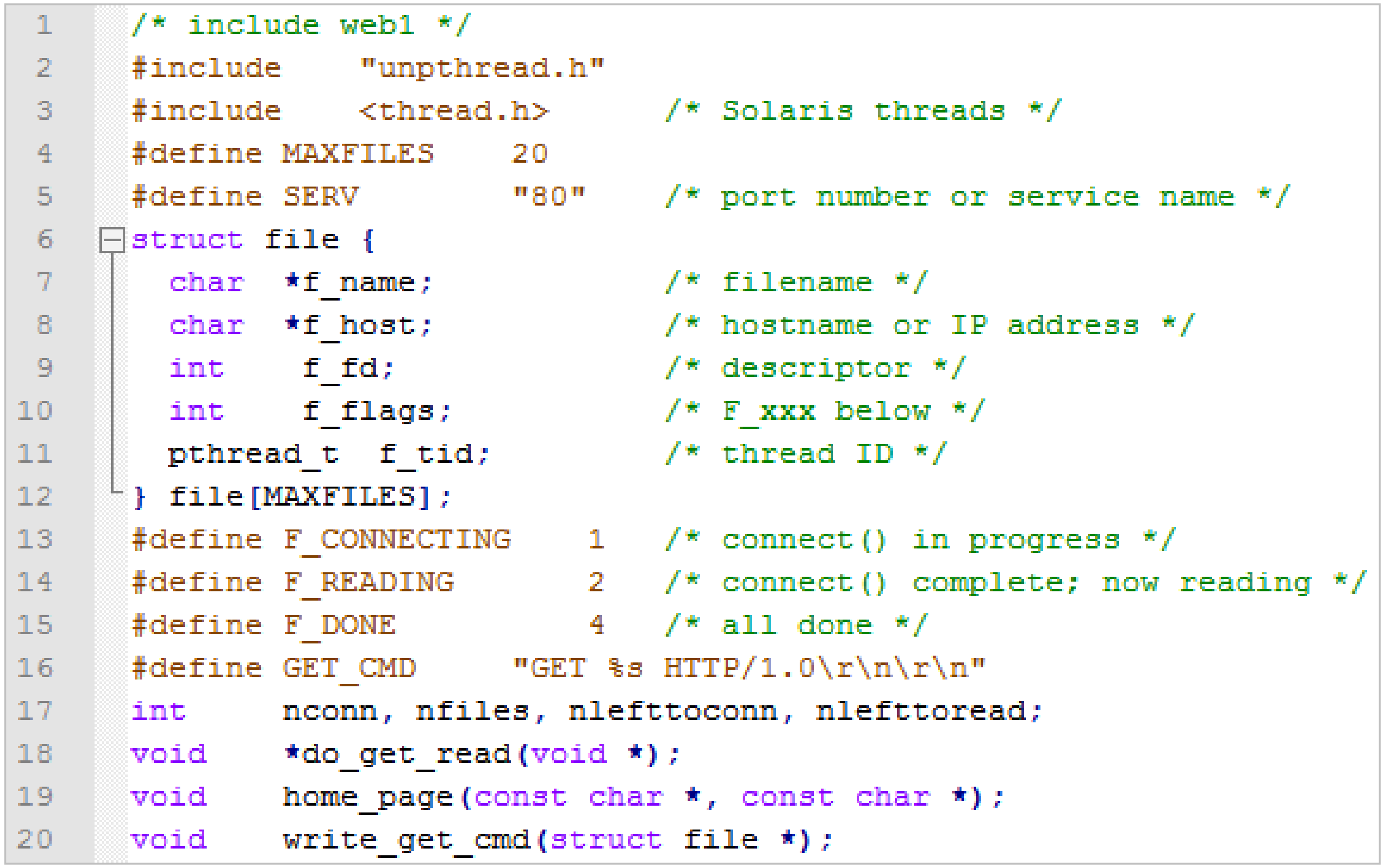
- This program will read up to 20 files from a Web server.
- We specify as command-line arguments
  - the maximum number of parallel connections,
  - the server's hostname, and
  - each of the filenames to fetch from the server.



```
bash$ web 3 www.foobar.com image1.gif image2.gif image3.gif image4.gif
    image5.gif image6.gif image7.gif
```

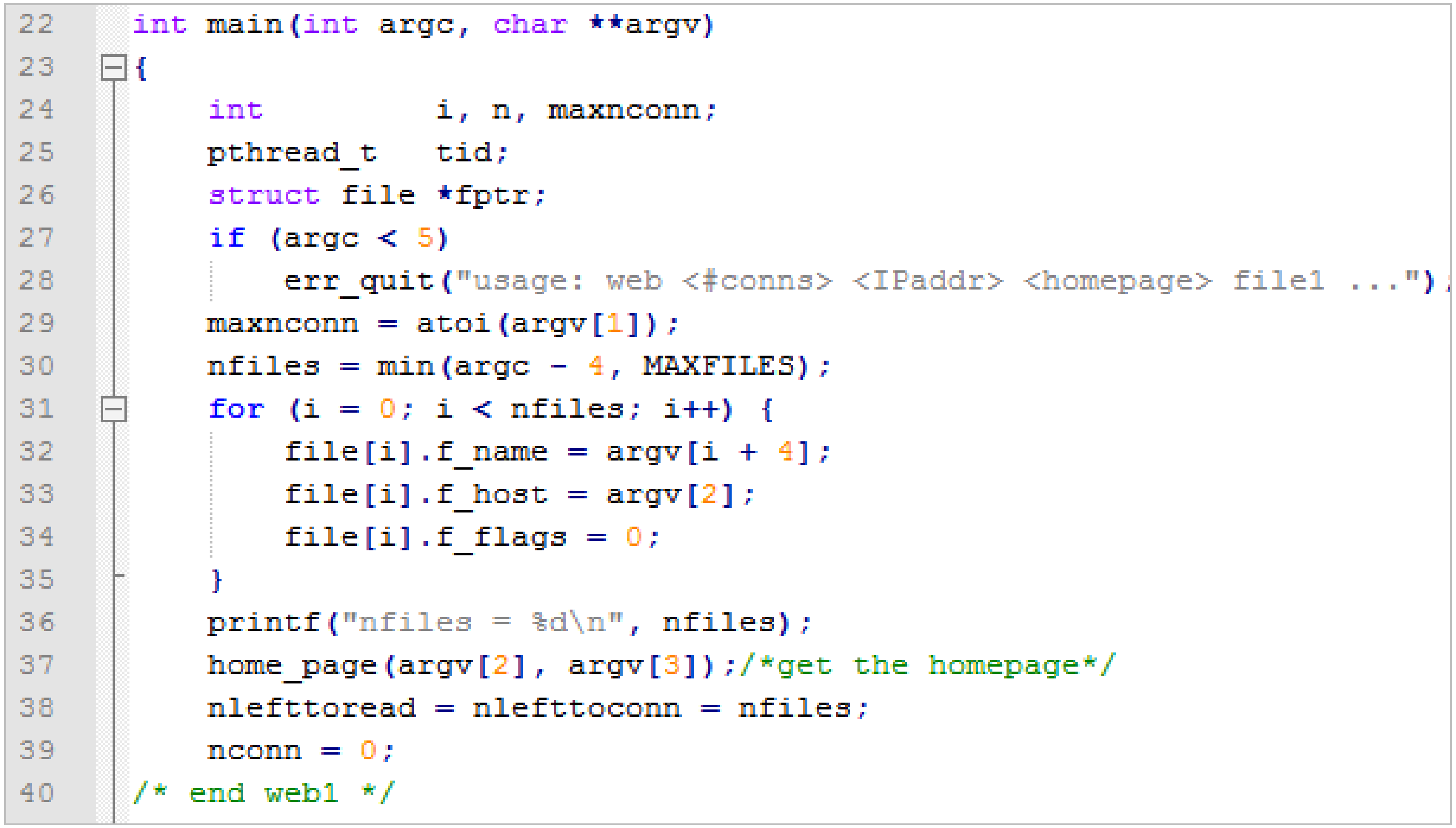
  - It means
    - three simultaneous connection
    - server’s hostname
    - filename for the home page
    - the files to be read
- T1: 26.6 & 26.9

Header file



```c
/* include web1 */
#include    "unpthread.h"
#include    <thread.h>      /* Solaris threads */
#define MAXFILES    20
#define SERV        "80"    /* port number or service name */
struct file {
  char   *f_name;           /* filename */
  char   *f_host;           /* hostname or IP address */
  int     f_fd;             /* descriptor */
  int     f_flags;          /* F_xxx below */
  pthread_t  f_tid;         /* thread ID */
} file[MAXFILES];
#define F_CONNECTING    1   /* connect() in progress */
#define F_READING       2   /* connect() complete; now reading */
#define F_DONE          4   /* all done */
#define GET_CMD     "GET %s HTTP/1.0\r\n\r\n"
int     nconn, nfiles, nlefttoconn, nlefttoread;
void    *do_get_read(void *);
void    home_page(const char *, const char *);
void    write_get_cmd(struct file *);
```

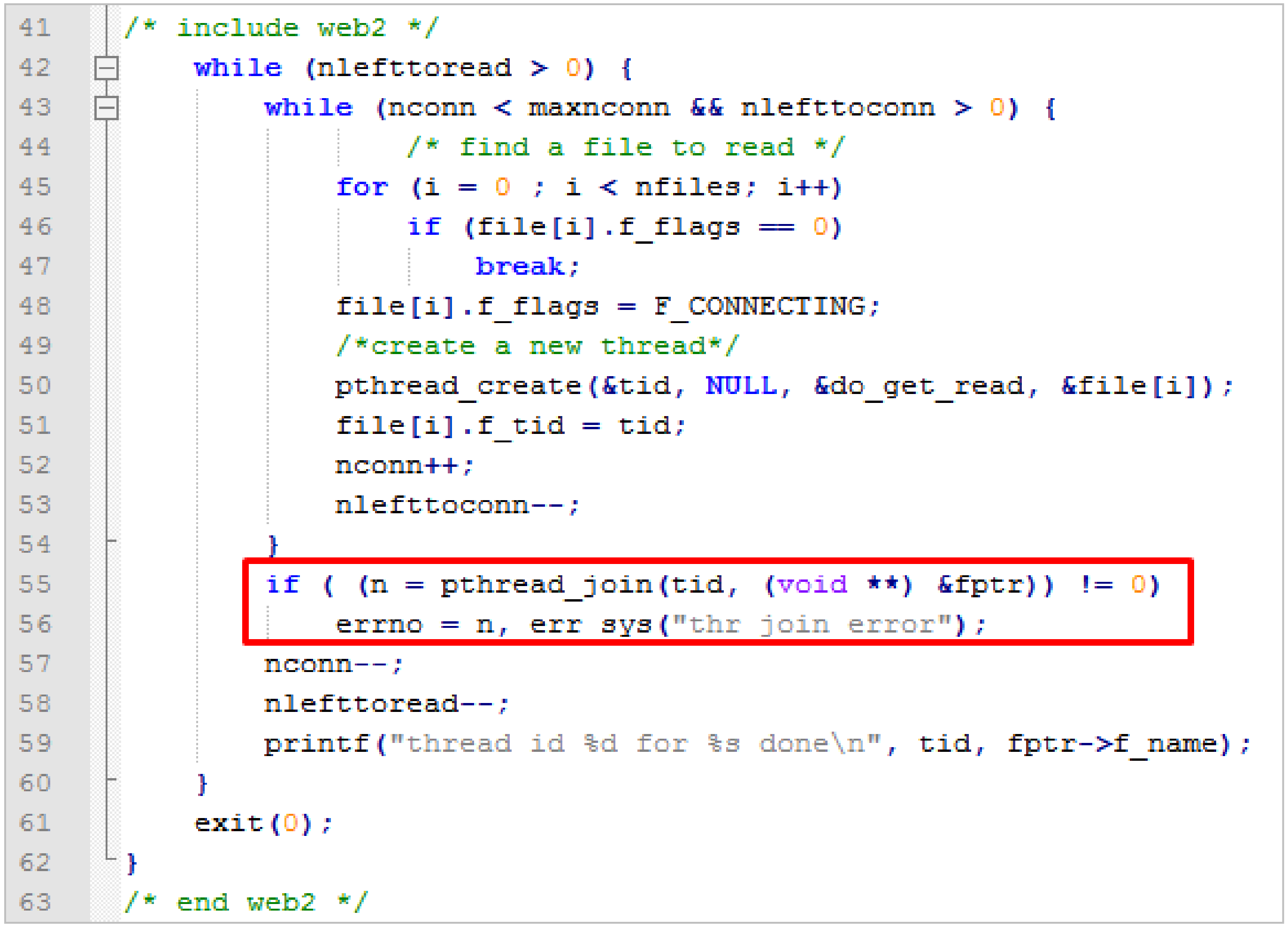
Main function



```c
int main(int argc, char **argv)
{
    int         i, n, maxnconn;
    pthread_t   tid;
    struct file *fptr;
    if (argc < 5)
        err_quit("usage: web <#conns> <IPaddr> <homepage> file1 ...");
    maxnconn = atoi(argv[1]);
    nfiles = min(argc - 4, MAXFILES);
    for (i = 0; i < nfiles; i++) {
        file[i].f_name = argv[i + 4];
        file[i].f_host = argv[2];
        file[i].f_flags = 0;
    }
    printf("nfiles = %d\n", nfiles);
    home_page(argv[2], argv[3]);/*get the homepage*/
    nlefttoread = nlefttoconn = nfiles;
    nconn = 0;
/* end web1 */
```

- Initialize structures

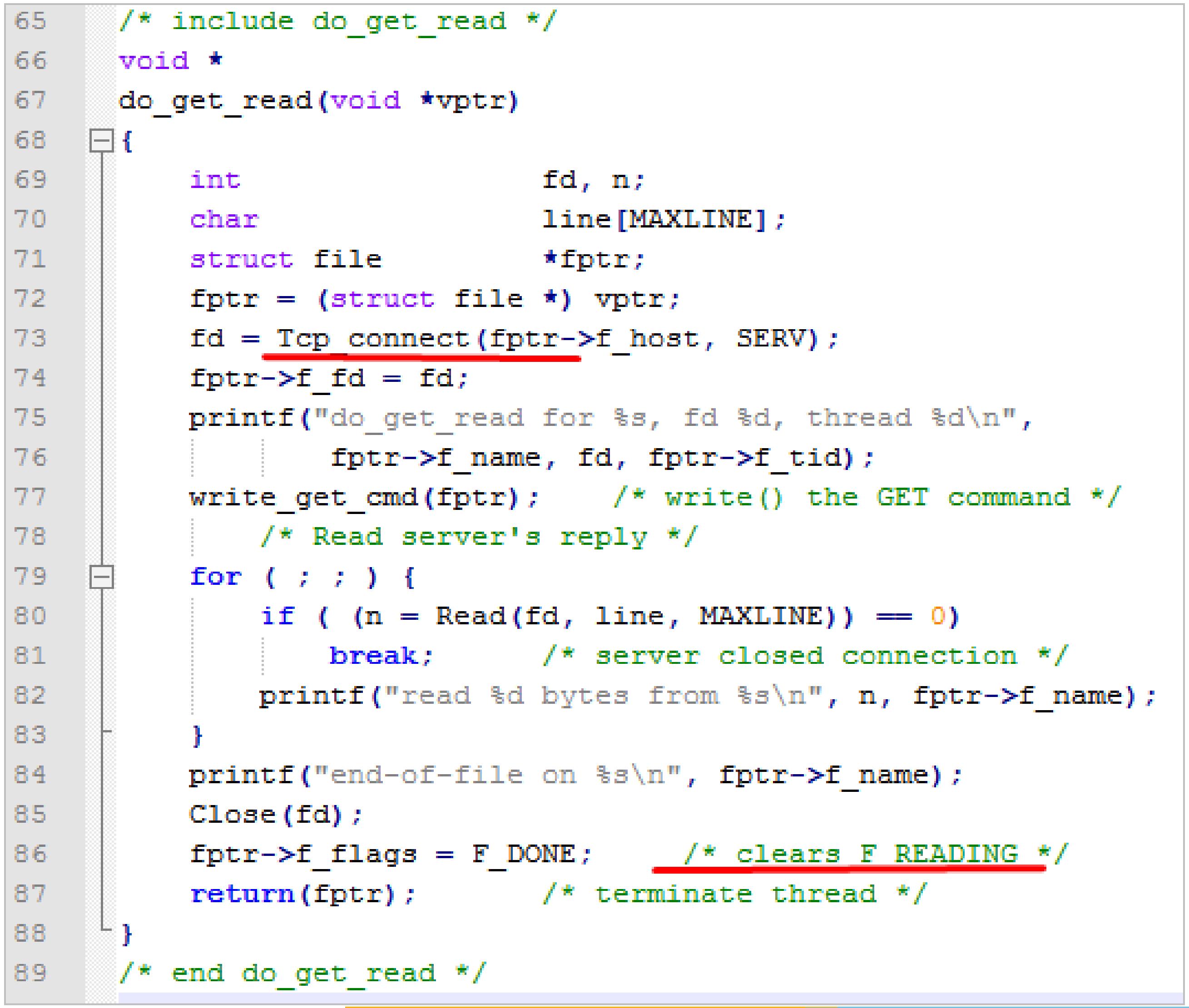
Main function



```c
/* include web2 */
    while (nlefttoread > 0) {
        while (nconn < maxnconn && nlefttoconn > 0) {
                /* find a file to read */
            for (i = 0 ; i < nfiles; i++)
                if (file[i].f_flags == 0)
                    break;
            file[i].f_flags = F_CONNECTING;
            /*create a new thread*/
            pthread_create(&tid, NULL, &do_get_read, &file[i]);
            file[i].f_tid = tid;
            nconn++;
            nlefttoconn--;
        }
        if ( (n = pthread_join(tid, (void **) &fptr)) != 0)
            errno = n, err_sys("thr join error");
        nconn--;
        nlefttoread--;
        printf("thread id %d for %s done\n", tid, fptr->f_name);
    }
    exit(0);
}
/* end web2 */
```

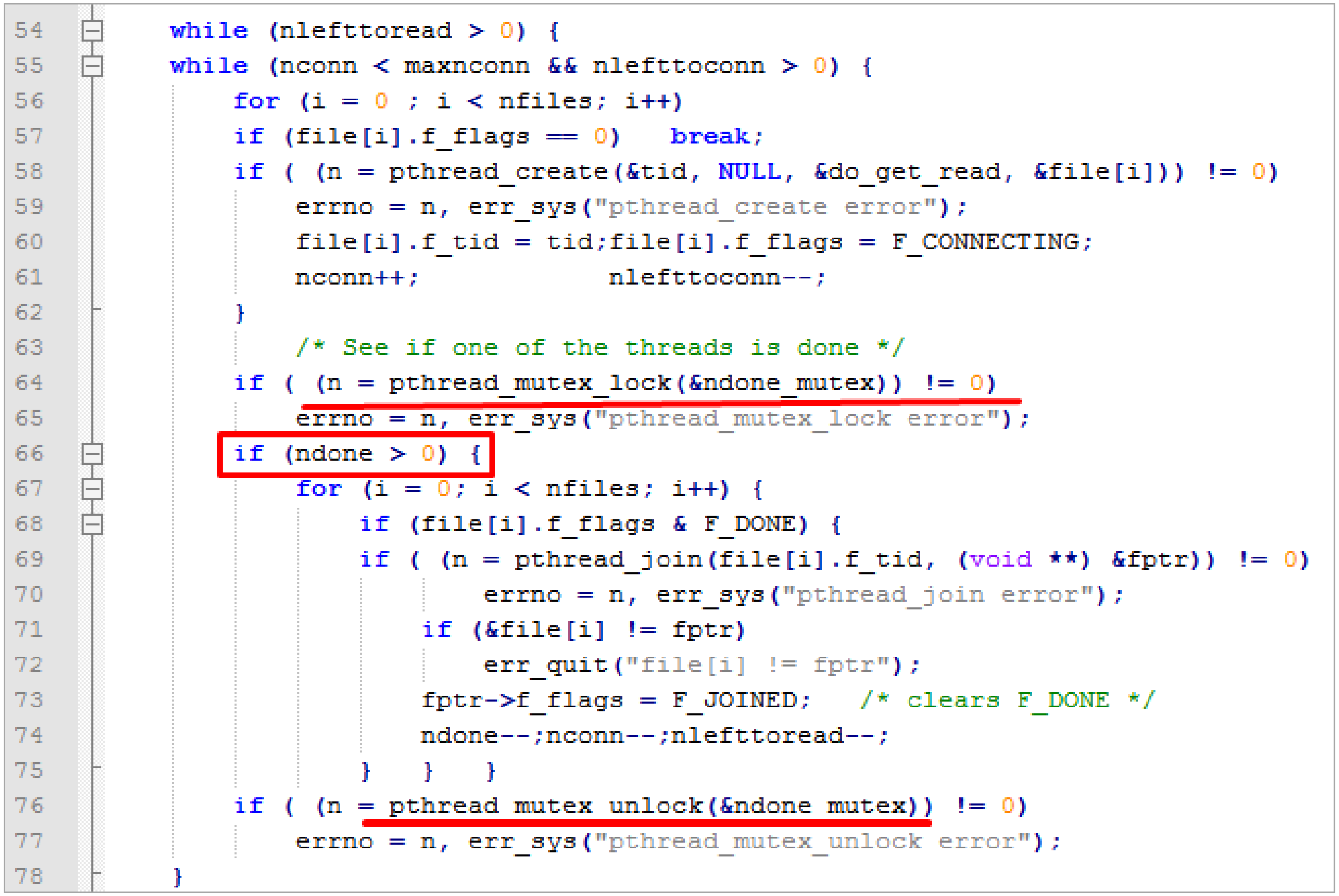
- Create maximum of maxconn threads and wait for them to terminate.

do_get_read function

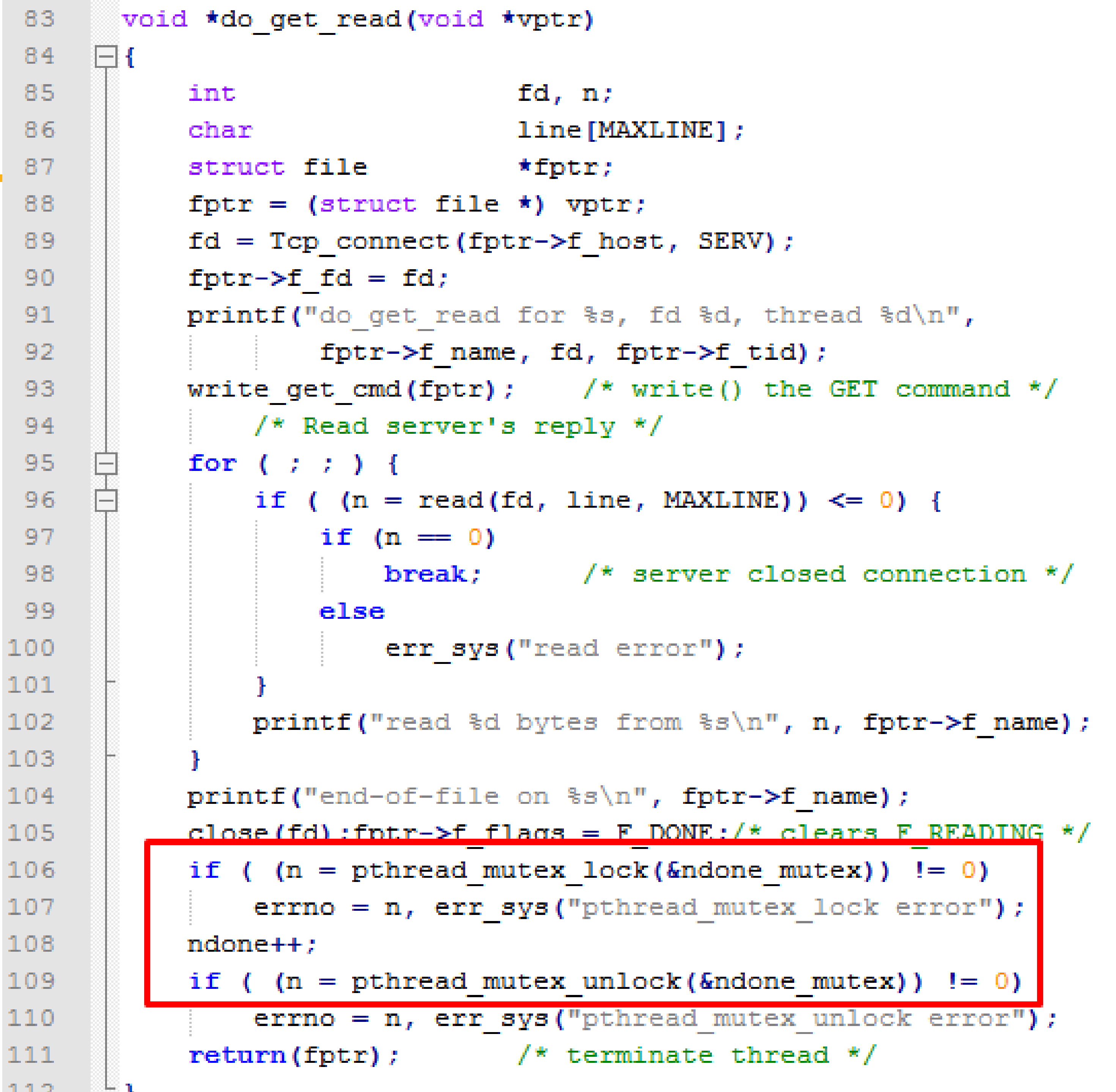


```
/* include do_get_read */
void *
do_get_read(void *vptr)
{
    int                     fd, n;
    char                    line[MAXLINE];
    struct file             *fptr;
    fptr = (struct file *) vptr;
    fd = Tcp_connect(fptr->f_host, SERV);
    fptr->f_fd = fd;
    printf("do_get_read for %s, fd %d, thread %d\n",
            fptr->f_name, fd, fptr->f_tid);
    write_get_cmd(fptr);    /* write() the GET command */
        /* Read server's reply */
    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;      /* server closed connection */
        printf("read %d bytes from %s\n", n, fptr->f_name);
    }
    printf("end-of-file on %s\n", fptr->f_name);
    Close(fd);
    fptr->f_flags = F_DONE;    /* clears F_READING */
    return(fptr);       /* terminate thread */
}
/* end do_get_read */
```

Polling for Available Threads

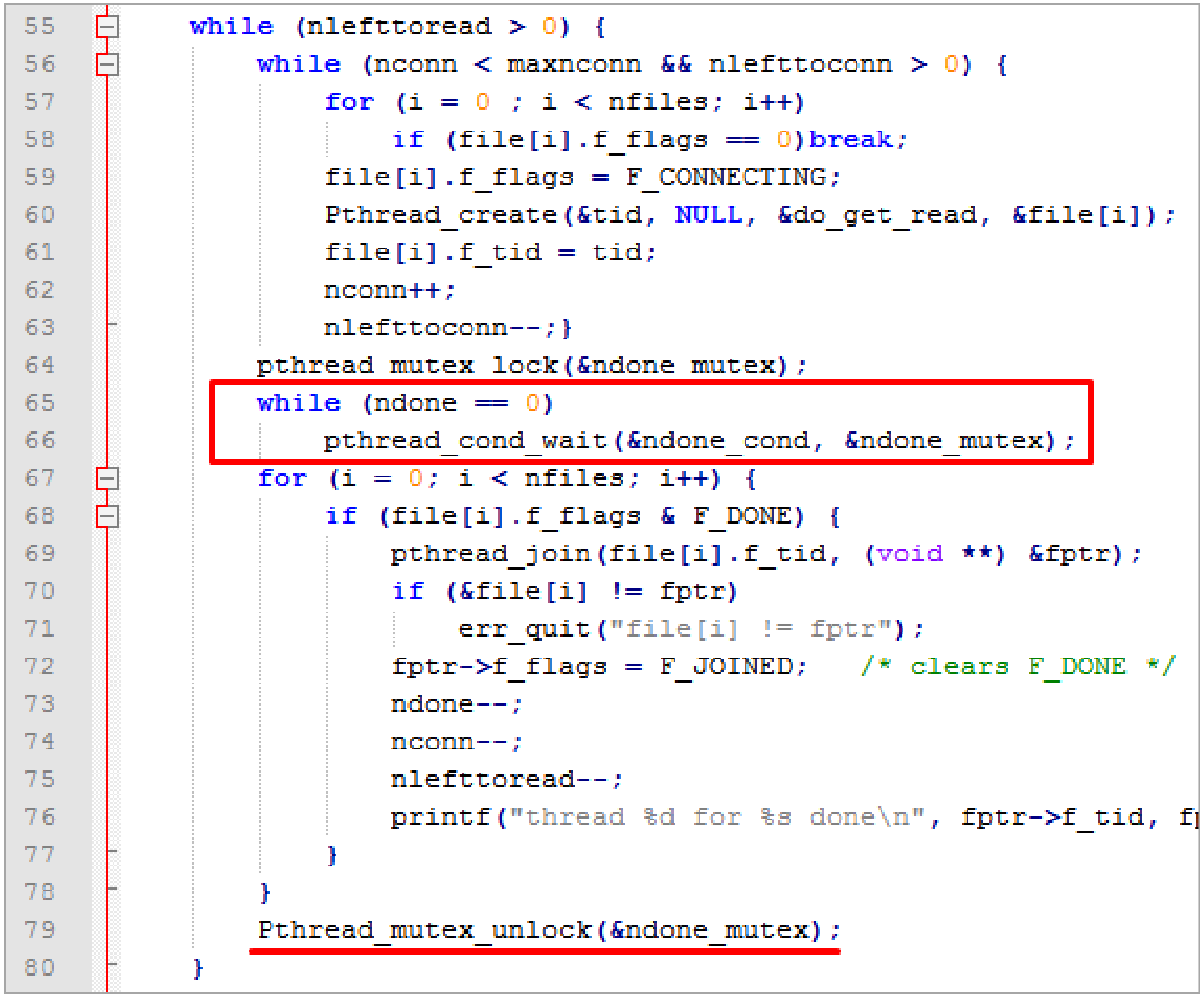


```
    while (nlefttoread > 0) {
    while (nconn < maxnconn && nlefttoconn > 0) {
        for (i = 0 ; i < nfiles; i++)
        if (file[i].f_flags == 0)   break;
        if ( (n = pthread_create(&tid, NULL, &do_get_read, &file[i])) != 0)
            errno = n, err_sys("pthread_create error");
            file[i].f_tid = tid;file[i].f_flags = F_CONNECTING;
            nconn++;            nlefttoconn--;
        }
            /* See if one of the threads is done */
        if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
            errno = n, err_sys("pthread_mutex_lock error");
        if (ndone > 0) {
            for (i = 0; i < nfiles; i++) {
                if (file[i].f_flags & F_DONE) {
                if ( (n = pthread_join(file[i].f_tid, (void **) &fptr)) != 0)
                        errno = n, err_sys("pthread_join error");
                    if (&file[i] != fptr)
                        err_quit("file[i] != fptr");
                    fptr->f_flags = F_JOINED;   /* clears F_DONE */
                    ndone--;nconn--;nlefttoread--;
                }   }   }
        if ( (n = pthread_mutex_unlock(&ndone_mutex)) != 0)
            errno = n, err_sys("pthread_mutex_unlock error");
    }
```



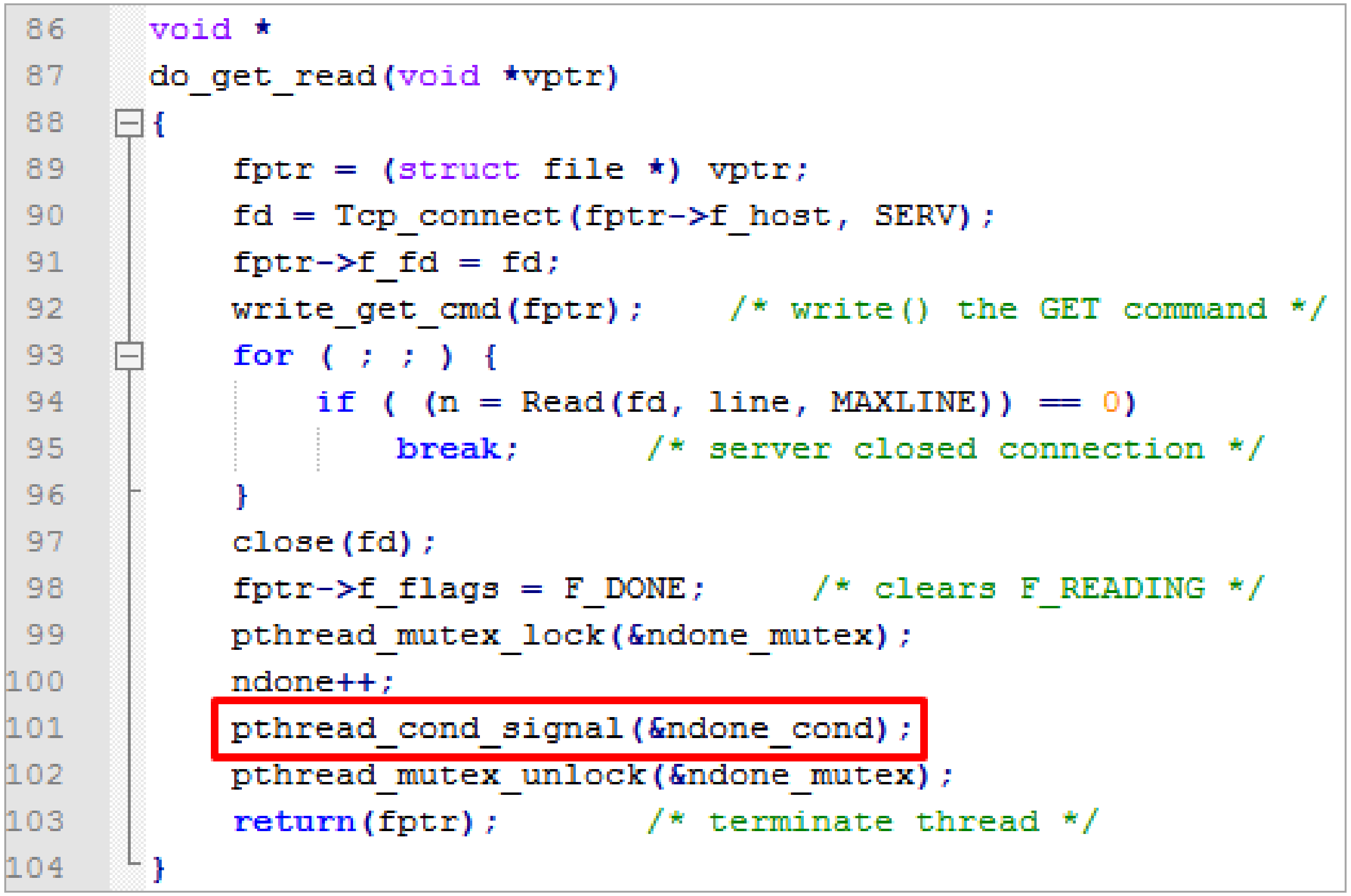
```c
void *do_get_read(void *vptr)
{
    int                     fd, n;
    char                    line[MAXLINE];
    struct file             *fptr;
    fptr = (struct file *) vptr;
    fd = Tcp_connect(fptr->f_host, SERV);
    fptr->f_fd = fd;
    printf("do_get_read for %s, fd %d, thread %d\n",
            fptr->f_name, fd, fptr->f_tid);
    write_get_cmd(fptr);    /* write() the GET command */
        /* Read server's reply */
    for ( ; ; ) {
        if ( (n = read(fd, line, MAXLINE)) <= 0) {
            if (n == 0)
                break;      /* server closed connection */
            else
                err_sys("read error");
        }
        printf("read %d bytes from %s\n", n, fptr->f_name);
    }
    printf("end-of-file on %s\n", fptr->f_name);
    close(fd);fptr->f_flags = F_DONE;/* clears F_READING */
    if ( (n = pthread_mutex_lock(&ndone_mutex)) != 0)
        errno = n, err_sys("pthread_mutex_lock error");
    ndone++;
    if ( (n = pthread_mutex_unlock(&ndone_mutex)) != 0)
        errno = n, err_sys("pthread_mutex_unlock error");
    return(fptr);       /* terminate thread */
}
```

Using Condition Variables



```
    while (nlefttoread > 0) {
        while (nconn < maxnconn && nlefttoconn > 0) {
            for (i = 0 ; i < nfiles; i++)
                if (file[i].f_flags == 0)break;
            file[i].f_flags = F_CONNECTING;
            Pthread_create(&tid, NULL, &do_get_read, &file[i]);
            file[i].f_tid = tid;
            nconn++;
            nlefttoconn--;}
        pthread_mutex_lock(&ndone_mutex);
        while (ndone == 0)
            pthread_cond_wait(&ndone_cond, &ndone_mutex);
        for (i = 0; i < nfiles; i++) {
            if (file[i].f_flags & F_DONE) {
                pthread_join(file[i].f_tid, (void **) &fptr);
                if (&file[i] != fptr)
                    err_quit("file[i] != fptr");
                fptr->f_flags = F_JOINED;   /* clears F_DONE */
                ndone--;
                nconn--;
                nlefttoread--;
                printf("thread %d for %s done\n", fptr->f_tid, f
            }
        }
        Pthread_mutex_unlock(&ndone_mutex);
    }
```

Using Condition Variables



```
void *
do_get_read(void *vptr)
{
    fptr = (struct file *) vptr;
    fd = Tcp_connect(fptr->f_host, SERV);
    fptr->f_fd = fd;
    write_get_cmd(fptr);     /* write() the GET command */
    for ( ; ; ) {
        if ( (n = Read(fd, line, MAXLINE)) == 0)
            break;       /* server closed connection */
    }
    close(fd);
    fptr->f_flags = F_DONE;      /* clears F_READING */
    pthread_mutex_lock(&ndone_mutex);
    ndone++;
    pthread_cond_signal(&ndone_cond);
    pthread_mutex_unlock(&ndone_mutex);
    return(fptr);        /* terminate thread */
}
```

Performance

- Table shows the clock time required to fetch a Web server's home page, followed by nine image files from that server.
  - The RTT to the server is about 150 ms.
  - The home page size was 4,017 bytes and the average size of the 9 image files was 1,621 bytes.
  - TCP's segment size was 512 bytes.
- Most of the improvement is obtained with three simultaneous connections.

# simultaneous connections	Clock time (seconds), non blocking	Clock time(secs) Threads
1	6.0	6.3
2	4.1	4.2
3	3.0	3.1
4	2.8	3.0
5	2.5	2.7
6	2.4	2.5
7	2.3	2.3
8	2.2	2.3
9	2.0	2.2

innovate achieve lead

BITS Pilani
Pilani Campus

Server Design Alternatives

T1: ch30

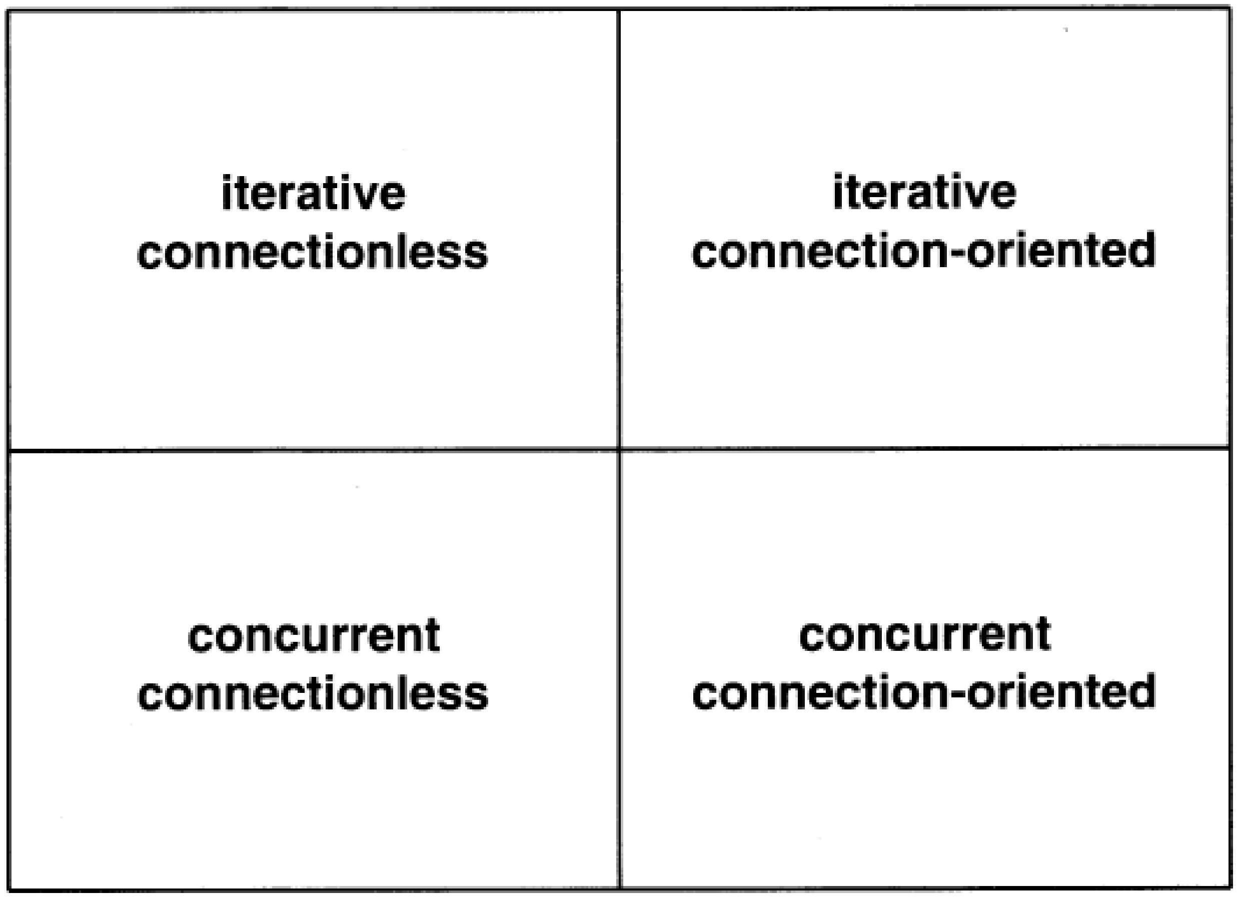
Iterative vs Concurrent

- Iterative Severs
  - Process one client at a time. Clients will experience significant delays in response.
  - Suitable when:
    - suitable only when client requests can be handled quickly, since each client must wait until all of the preceding clients have been serviced.
    - A typical scenario: client and server exchange a single request and response.
- Concurrent Servers
  - Handle multiple clients simultaneously. Can take advantage of CPU-IO overlap.
  - Suitable when:
    - Concurrent servers are suitable when a significant amount of processing time is required to handle each request
    - Or where the client and server engage in an extended conversation, passing messages back and forth.

Stateless Vs. Stateful Servers

- Information that a server maintains about the status of ongoing interactions with clients is called state information.
- Servers that do not keep any state information are called stateless servers; others are called stateful servers.
- Stateful Servers:
  - Keeping a small amount of information in a server
    - can reduce the size of messages
    - can allow the server to respond to requests quickly.
  - Server can compute an incremental response as each new request arrives.
- State information in a server can become incorrect if
  - messages are lost, duplicated, or delivered out of order, or if the client computer crashes and reboots.

General Server Categories



iterative connectionless	iterative connection-oriented
concurrent connectionless	concurrent connection-oriented

General Server Categories

- Iterative Connectionless
  - Common form of connectionless server.
  - Often stateless.
  - Used when it requires trivial amount of processing for each request.
- Iterative, Connection-Oriented Server
  - A less common server type
  - used for services that require a trivial amount of processing for each request, but for which reliable transport is necessary.
  - Because the overhead associated with establishing and terminating connections can be high, the average response time can be non-trivial.

General Server Categories

- Concurrent, Connectionless Server
  - An uncommon type
  - The server creates a new process to handle each request.
    - On many systems, the added cost of process creation dominates the added efficiency gained from concurrency.
  - To justify concurrency,
    - either the time required to create a new process must be significantly less than the time required to compute a response
    - or concurrent requests must be able to use many I/O devices simultaneously.

General Server Categories

- Concurrent, Connection-Oriented Server
  - The most general type of server
  - it offers reliable transport (i.e., it can be used across a wide area internet) as well as the ability to handle multiple requests concurrently.
  - Two basic implementations exist
    - concurrent processes/threads to handle multiple connections.
    - a single process and asynchronous I/O to handle multiple connections.

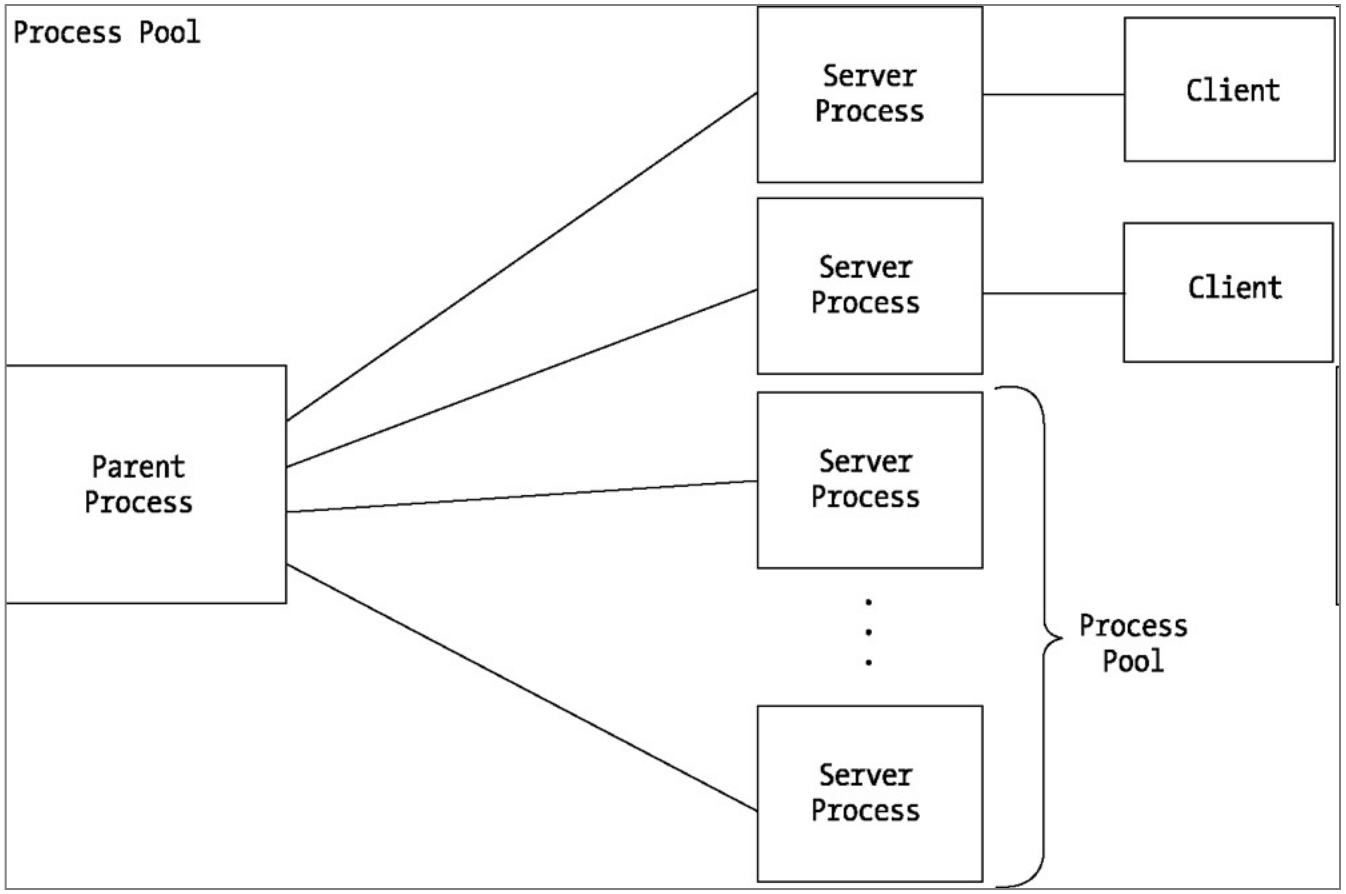
Preforked and prethreaded servers

- Traditional concurrent server model:
  - Fork a child after accepting a new client connection.
  - Good enough for low traffic services.
- For very high-load servers
  - web servers handling thousands of requests per minute
  - the cost of creating a new child (or even thread) for each client imposes a significant burden on the server.
- Instead of creating a new child process (or thread) for each client, the server precreates a fixed number of child processes (or threads) on startup.
  - Each child (thread) handles a new client. After completing one client, it accepts another connection.

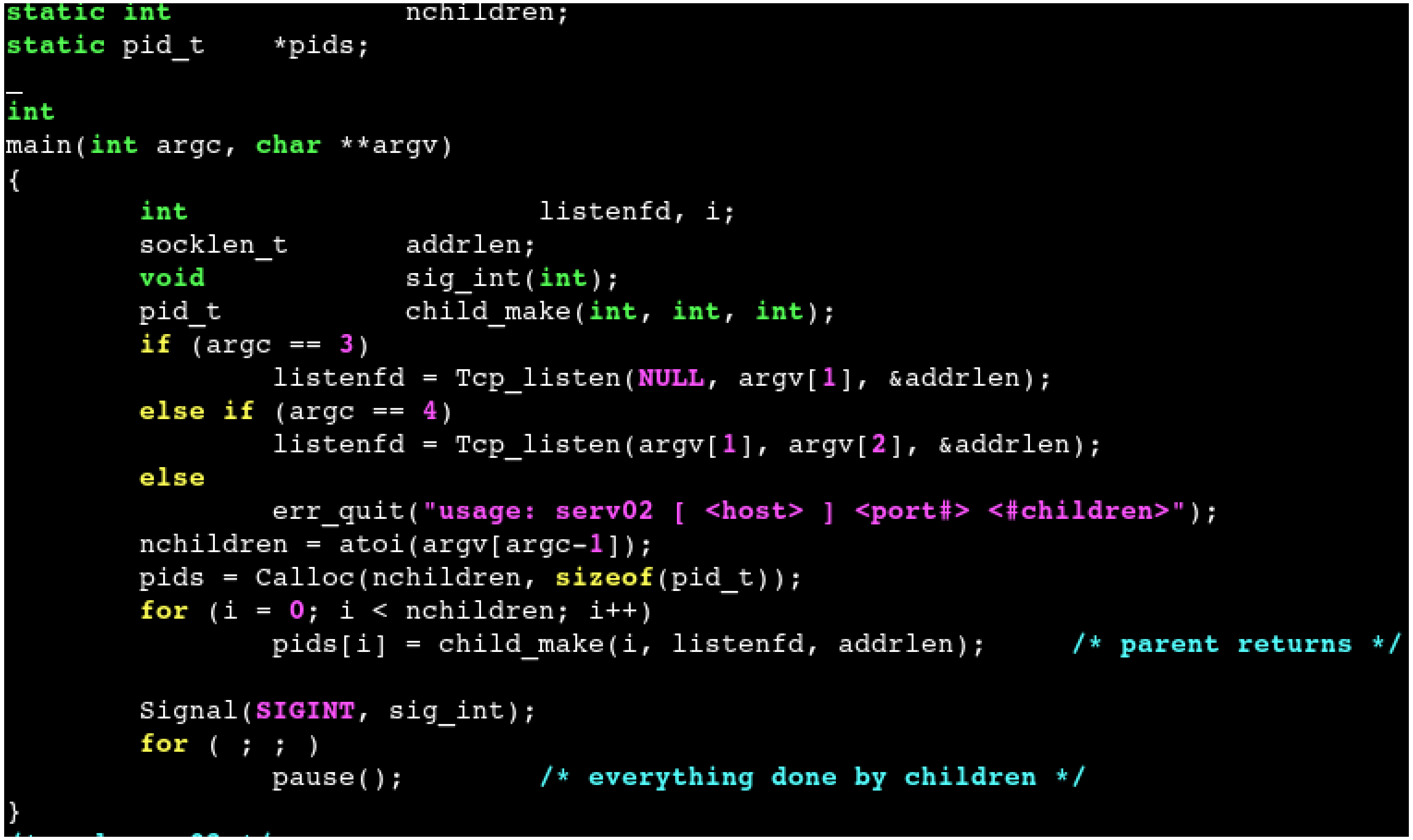
Preforking

- Different models
  - Child calls accept()
    - TCP Preforked Server, No Locking Around accept
    - TCP Preforked Server, Thread Locking Around accept
  - Parent calls accept() and passes the descriptor to child
    - TCP Preforked Server, Descriptor Passing

Preforking or Process Pool

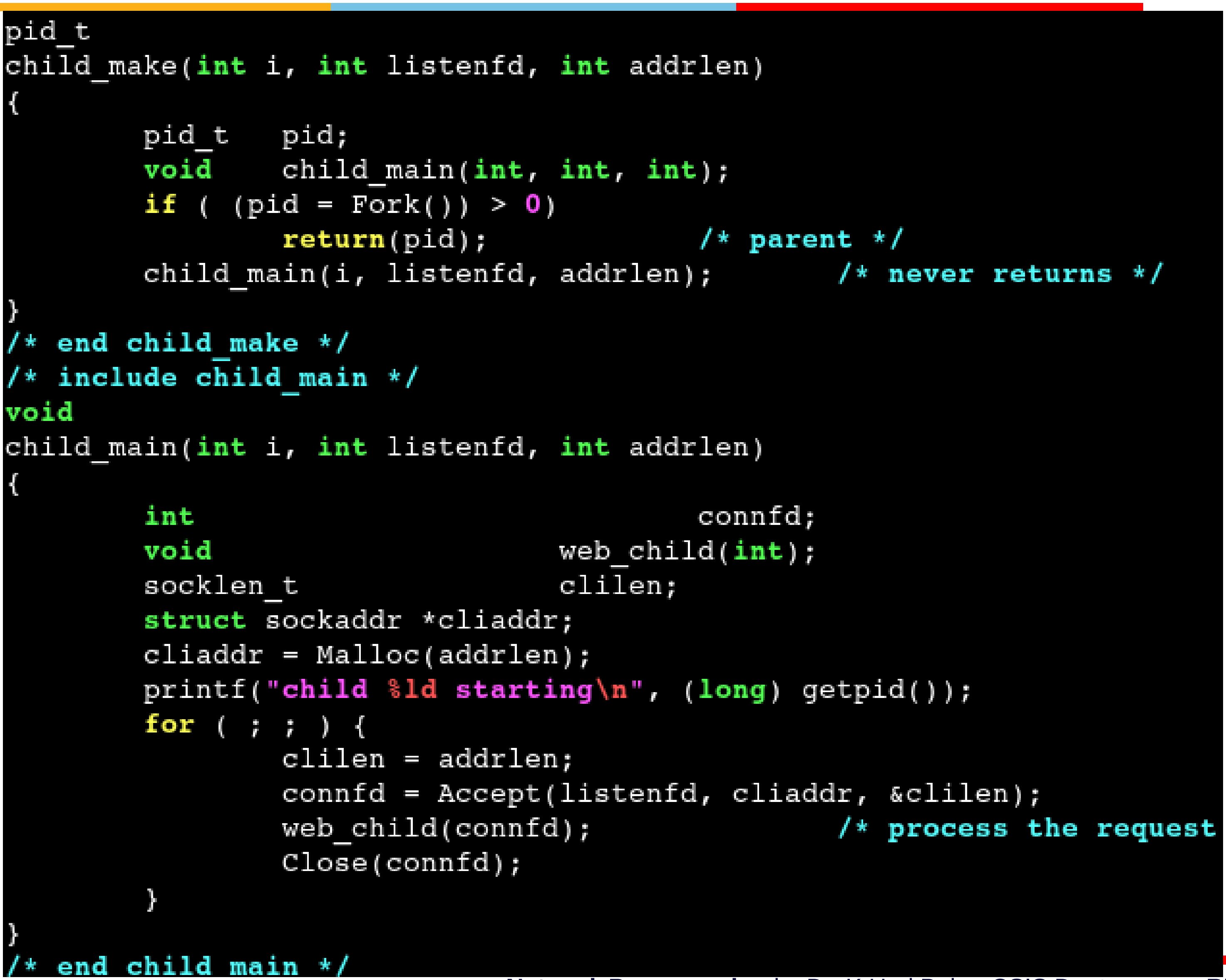


Preforking or Process Pool



```
static int              nchildren;
static pid_t    *pids;

_
int
main(int argc, char **argv)
{
        int                     listenfd, i;
        socklen_t       addrlen;
        void            sig_int(int);
        pid_t           child_make(int, int, int);
        if (argc == 3)
                listenfd = Tcp_listen(NULL, argv[1], &addrlen);
        else if (argc == 4)
                listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
        else
                err_quit("usage: serv02 [ <host> ] <port#> <#children>");
        nchildren = atoi(argv[argc-1]);
        pids = Calloc(nchildren, sizeof(pid_t));
        for (i = 0; i < nchildren; i++)
                pids[i] = child_make(i, listenfd, addrlen);     /* parent returns */

        Signal(SIGINT, sig_int);
        for ( ; ; )
                pause();        /* everything done by children */
}
```

Preforking or Process Pool: No Locking Around Accept

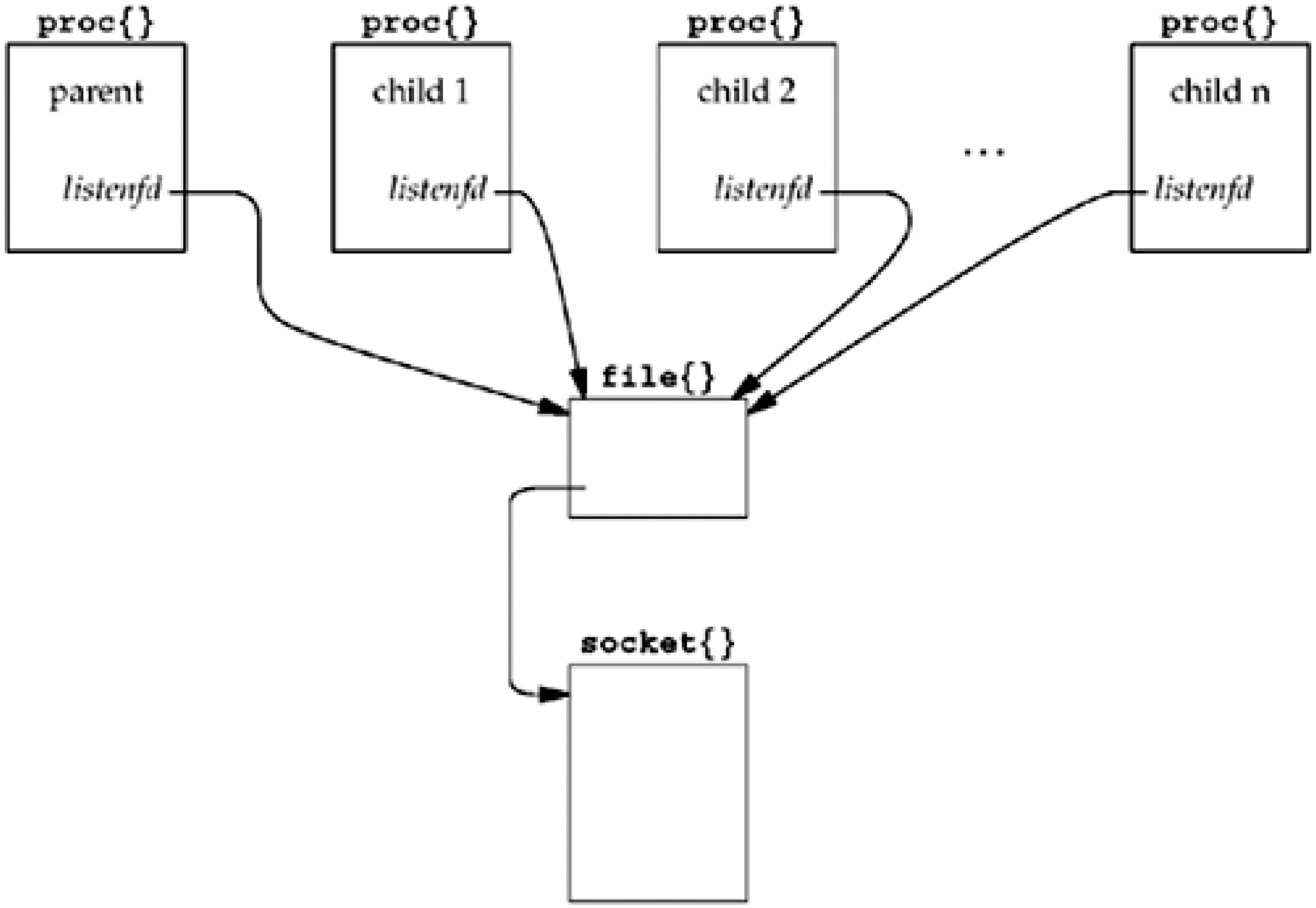


```
pid_t
child_make(int i, int listenfd, int addrlen)
{
        pid_t   pid;
        void    child_main(int, int, int);
        if ( (pid = Fork()) > 0)
                return(pid);                /* parent */
        child_main(i, listenfd, addrlen);       /* never returns */
}
/* end child_make */
/* include child_main */
void
child_main(int i, int listenfd, int addrlen)
{
        int                             connfd;
        void                    web_child(int);
        socklen_t               clilen;
        struct sockaddr *cliaddr;
        cliaddr = Malloc(addrlen);
        printf("child %ld starting\n", (long) getpid());
        for ( ; ; ) {
                clilen = addrlen;
                connfd = Accept(listenfd, cliaddr, &clilen);
                web_child(connfd);                      /* process the request
                Close(connfd);
        }
}
/* end child_main */
```

Preforking or Process Pool: No Locking Around Accept

- Advantages:
  - No cost of fork() before responding to client.
  - Process control is simpler.
- Disadvantages:
  - Parent must guess how many children to fork.
  - If too less, clients will experience delays in response.
  - If too excessive, system performance degrades.

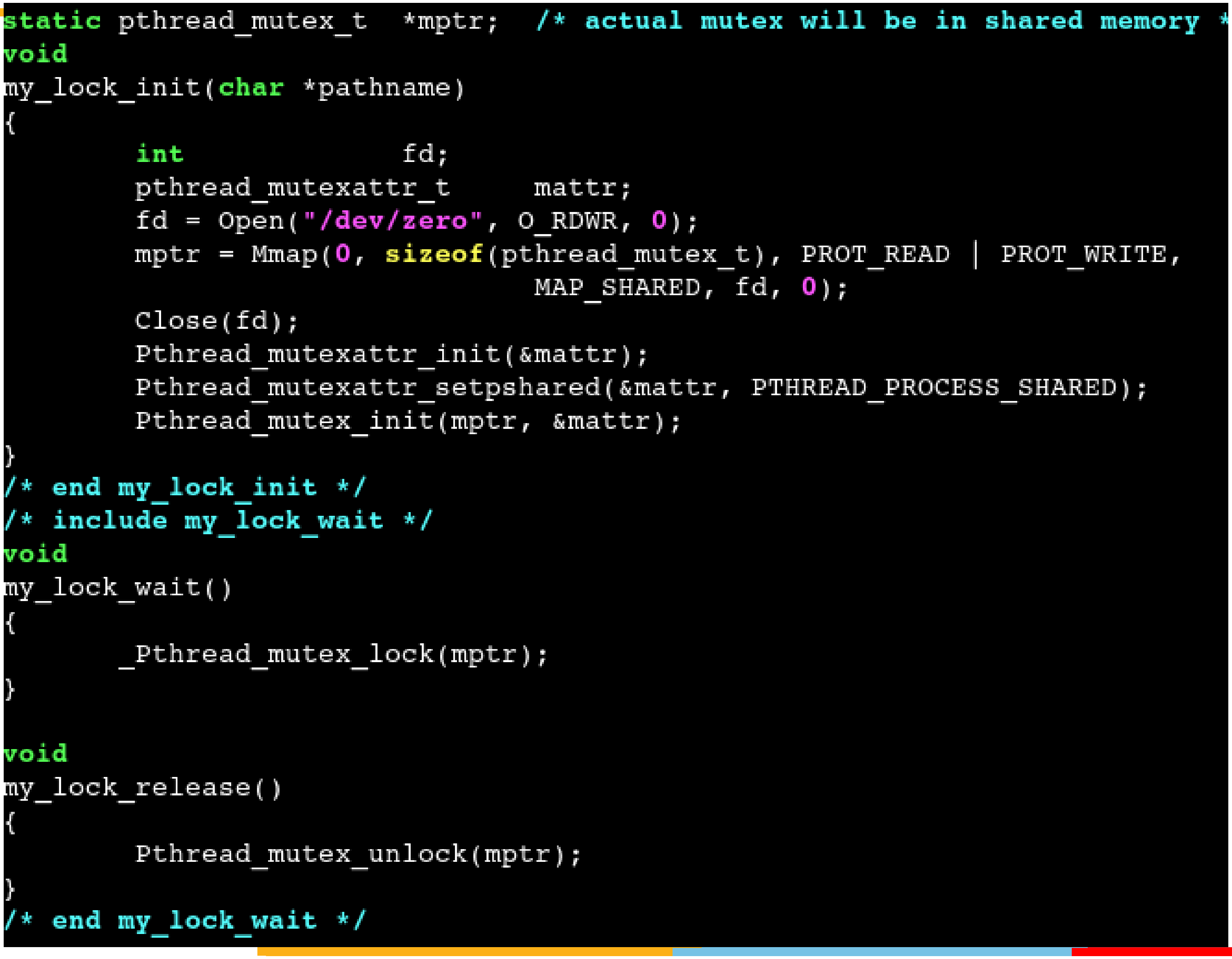
Thundering Herd Problem



Thundering Herd Problem

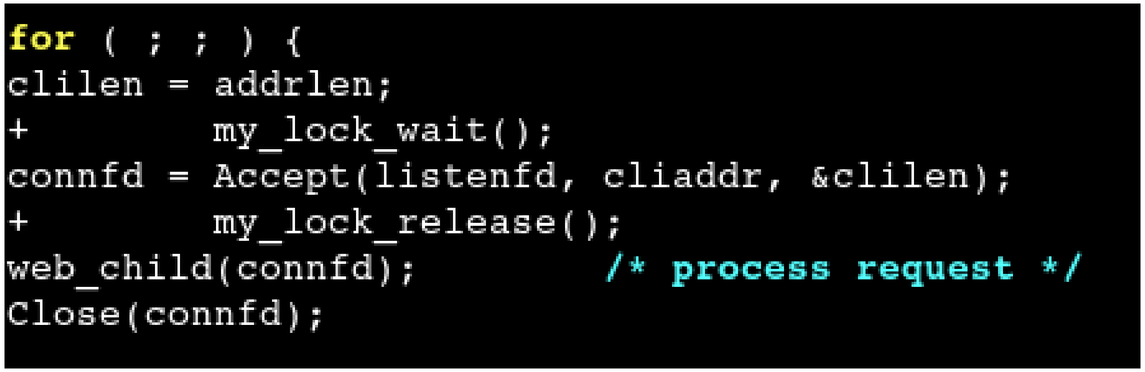
- When the program starts, N children are created, and all N call accept and all are put to sleep by the kernel.
- When the first client connection arrives, all N children are awakened.
  - because all N have gone to sleep on the same "wait channel“ because all N share the same listening descriptor.
- Even though all N are awakened, the first of the N to run will obtain the connection and the remaining N - 1 will all go back to sleep.

Preforking: Locking Around accept



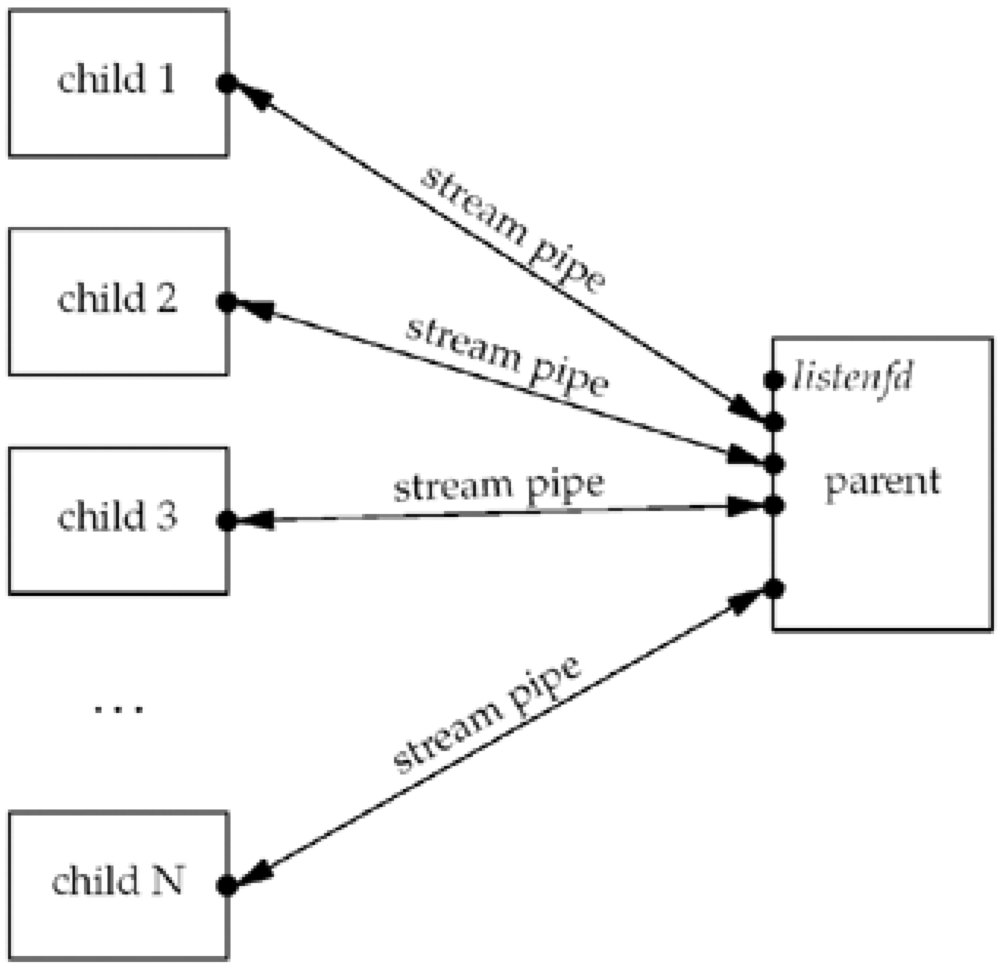
```
static pthread_mutex_t  *mptr;  /* actual mutex will be in shared memory *
void
my_lock_init(char *pathname)
{
        int                 fd;
        pthread_mutexattr_t     mattr;
        fd = Open("/dev/zero", O_RDWR, 0);
        mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);
        Close(fd);
        Pthread_mutexattr_init(&mattr);
        Pthread_mutexattr_setpshared(&mattr, PTHREAD_PROCESS_SHARED);
        Pthread_mutex_init(mptr, &mattr);
}
/* end my_lock_init */
/* include my_lock_wait */
void
my_lock_wait()
{
       _Pthread_mutex_lock(mptr);
}

void
my_lock_release()
{
        Pthread_mutex_unlock(mptr);
}
/* end my_lock_wait */
```

Preforking: Locking Around accept

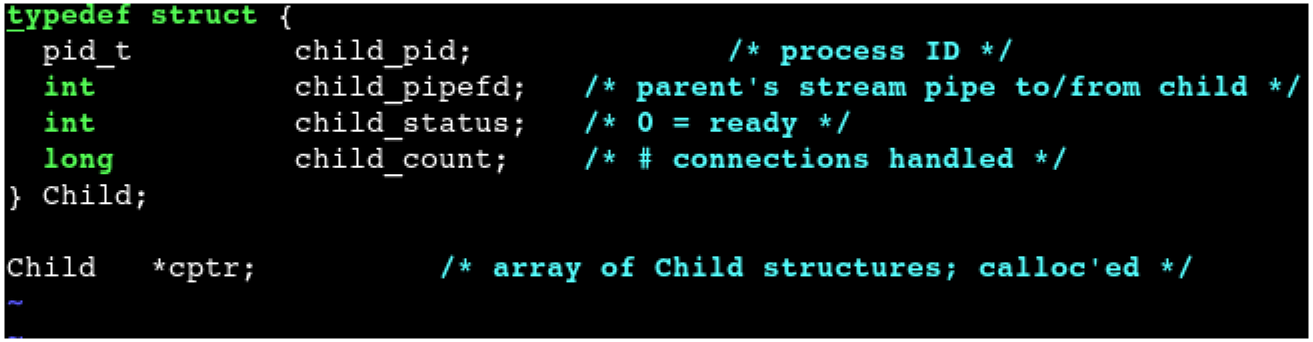


```
for ( ; ; ) {
clilen = addrlen;
+          my_lock_wait();
connfd = Accept(listenfd, cliaddr, &clilen);
+          my_lock_release();
web_child(connfd);         /* process request */
Close(connfd);
```

Preforking: Descriptor Passing



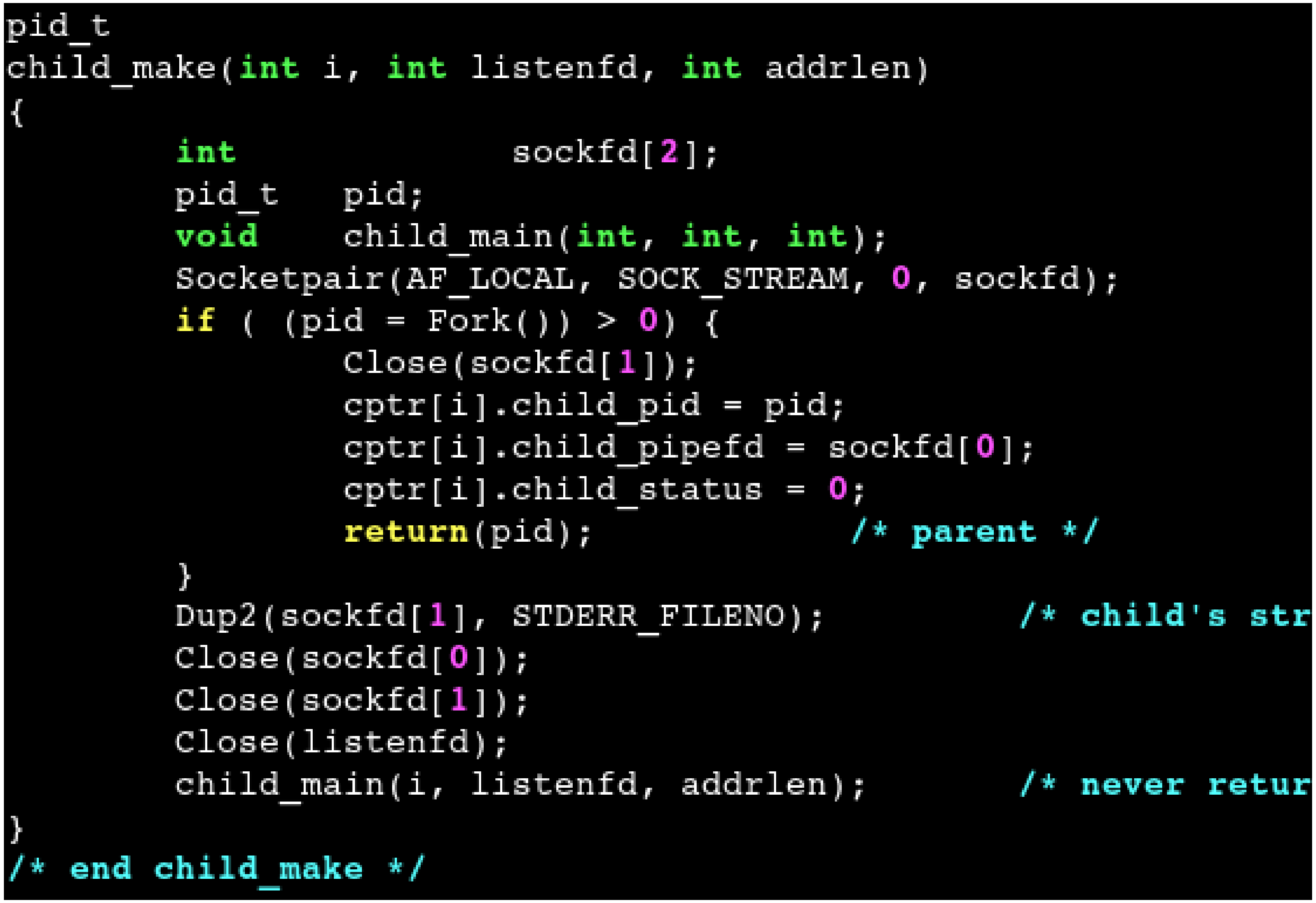
Preforking: Descriptor Passing



```
typedef struct {
  pid_t          child_pid;              /* process ID */
  int            child_pipefd;   /* parent's stream pipe to/from child */
  int            child_status;   /* 0 = ready */
  long           child_count;    /* # connections handled */
} Child;

Child   *cptr;          /* array of Child structures; calloc'ed */
~
```

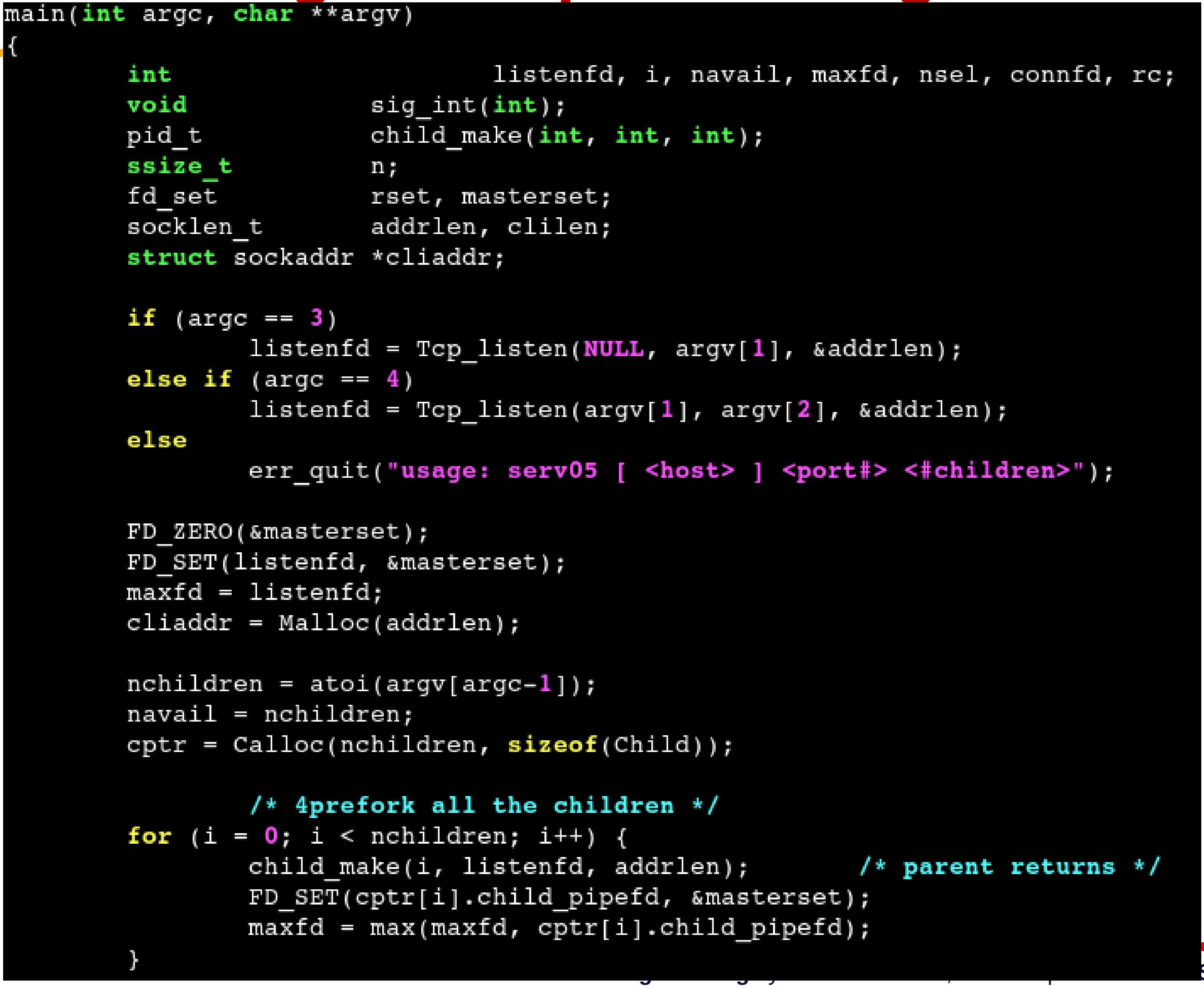
- Parent maintains this structure for each child.

Preforking: Descriptor Passing



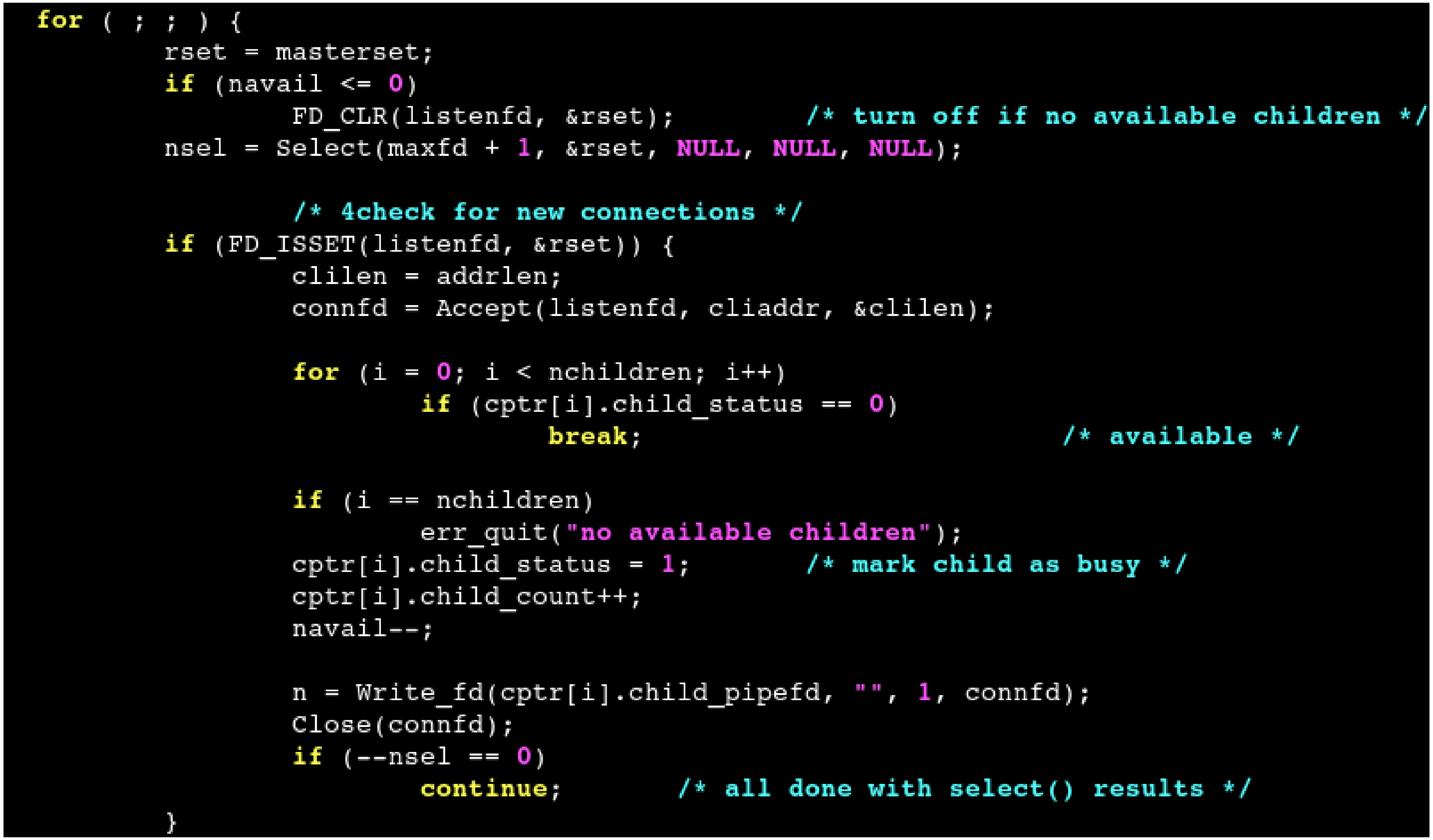
```
pid_t
child_make(int i, int listenfd, int addrlen)
{
        int                sockfd[2];
        pid_t   pid;
        void    child_main(int, int, int);
        Socketpair(AF_LOCAL, SOCK_STREAM, 0, sockfd);
        if ( (pid = Fork()) > 0) {
                Close(sockfd[1]);
                cptr[i].child_pid = pid;
                cptr[i].child_pipefd = sockfd[0];
                cptr[i].child_status = 0;
                return(pid);            /* parent */
        }
        Dup2(sockfd[1], STDERR_FILENO);         /* child's str
        Close(sockfd[0]);
        Close(sockfd[1]);
        Close(listenfd);
        child_main(i, listenfd, addrlen);       /* never retur
}
/* end child_make */
```

Preforking: Descriptor Passing



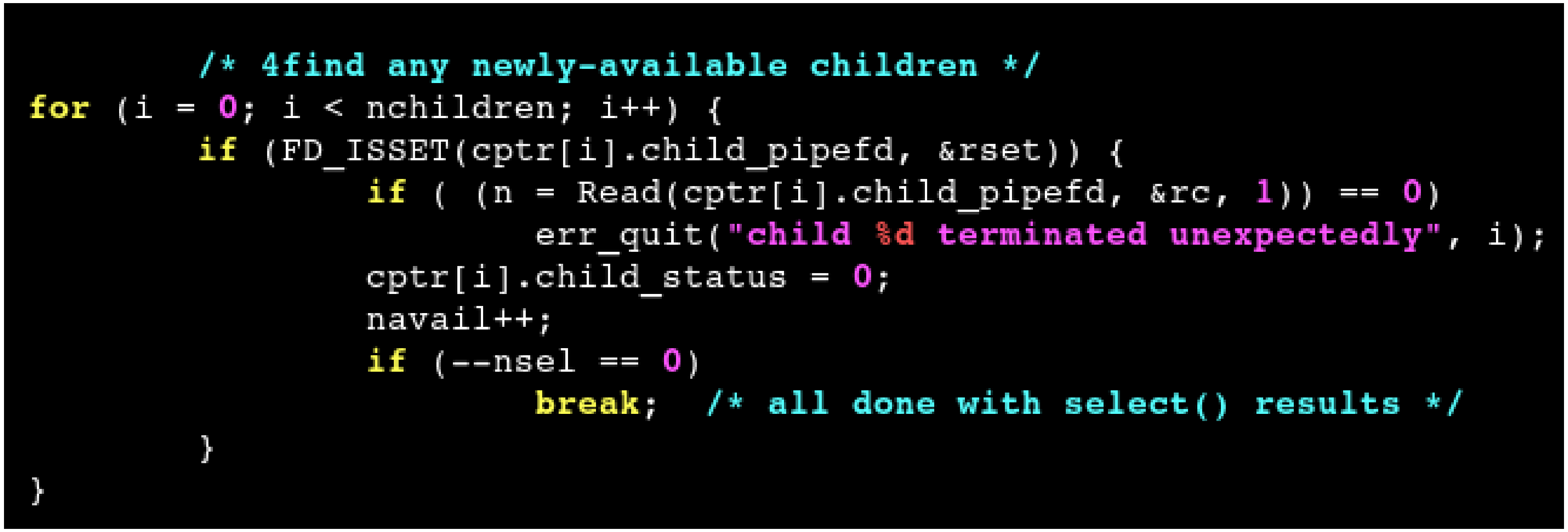
```
main(int argc, char **argv)
{
        int                     listenfd, i, navail, maxfd, nsel, connfd, rc;
        void            sig_int(int);
        pid_t           child_make(int, int, int);
        ssize_t         n;
        fd_set          rset, masterset;
        socklen_t       addrlen, clilen;
        struct sockaddr *cliaddr;

        if (argc == 3)
                listenfd = Tcp_listen(NULL, argv[1], &addrlen);
        else if (argc == 4)
                listenfd = Tcp_listen(argv[1], argv[2], &addrlen);
        else
                err_quit("usage: serv05 [ <host> ] <port#> <#children>");

        FD_ZERO(&masterset);
        FD_SET(listenfd, &masterset);
        maxfd = listenfd;
        cliaddr = Malloc(addrlen);

        nchildren = atoi(argv[argc-1]);
        navail = nchildren;
        cptr = Calloc(nchildren, sizeof(Child));

                /* 4prefork all the children */
        for (i = 0; i < nchildren; i++) {
                child_make(i, listenfd, addrlen);       /* parent returns */
                FD_SET(cptr[i].child_pipefd, &masterset);
                maxfd = max(maxfd, cptr[i].child_pipefd);
        }
```

Descriptor Passing



```
for ( ; ; ) {
        rset = masterset;
        if (navail <= 0)
                FD_CLR(listenfd, &rset);        /* turn off if no available children */
        nsel = Select(maxfd + 1, &rset, NULL, NULL, NULL);

                /* 4check for new connections */
        if (FD_ISSET(listenfd, &rset)) {
                clilen = addrlen;
                connfd = Accept(listenfd, cliaddr, &clilen);

                for (i = 0; i < nchildren; i++)
                        if (cptr[i].child_status == 0)
                                break;                          /* available */

                if (i == nchildren)
                        err_quit("no available children");
                cptr[i].child_status = 1;       /* mark child as busy */
                cptr[i].child_count++;
                navail--;

                n = Write_fd(cptr[i].child_pipefd, "", 1, connfd);
                Close(connfd);
                if (--nsel == 0)
                        continue;       /* all done with select() results */
        }
```

Descriptor Passing



```
        /* 4find any newly-available children */
for (i = 0; i < nchildren; i++) {
        if (FD_ISSET(cptr[i].child_pipefd, &rset)) {
                if ( (n = Read(cptr[i].child_pipefd, &rc, 1)) == 0)
                        err_quit("child %d terminated unexpectedly", i);
                cptr[i].child_status = 0;
                navail++;
                if (--nsel == 0)
                        break;  /* all done with select() results */
        }
}
```

Acknowledgements

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You