Network Programming

K Hari Babu
Department of Computer Science & Information Systems

BITS Pilani
Pilani Campus

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

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

```
#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);
 }
 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

```
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 */
```

```
/*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);
}
```

innovate achieve lead

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

```
/* 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

```c
/* 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

```c
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");
}
```

```
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

innovate achieve lead

- 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.

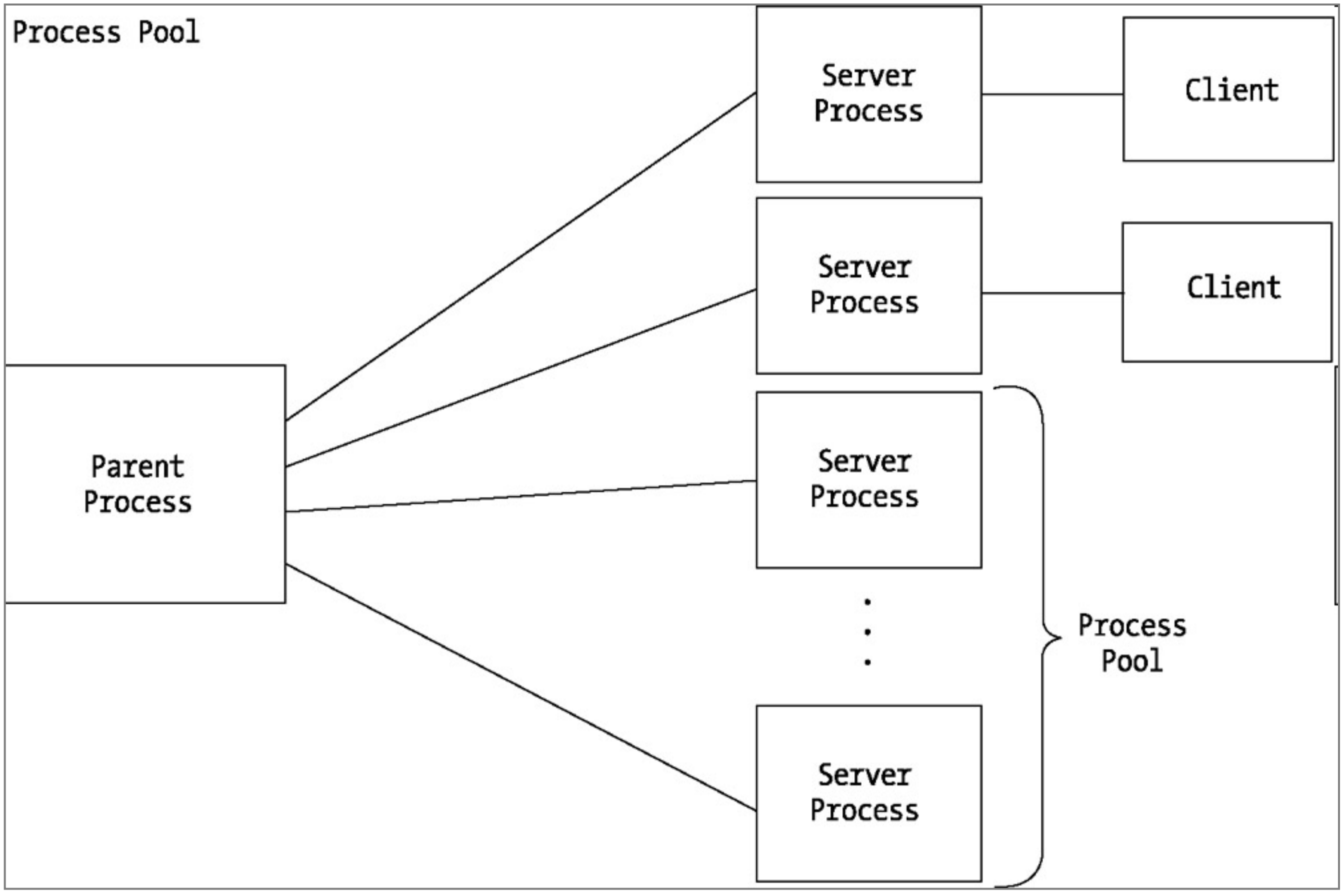
Preforked and prethreaded servers

- 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

```c
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 */
}
```

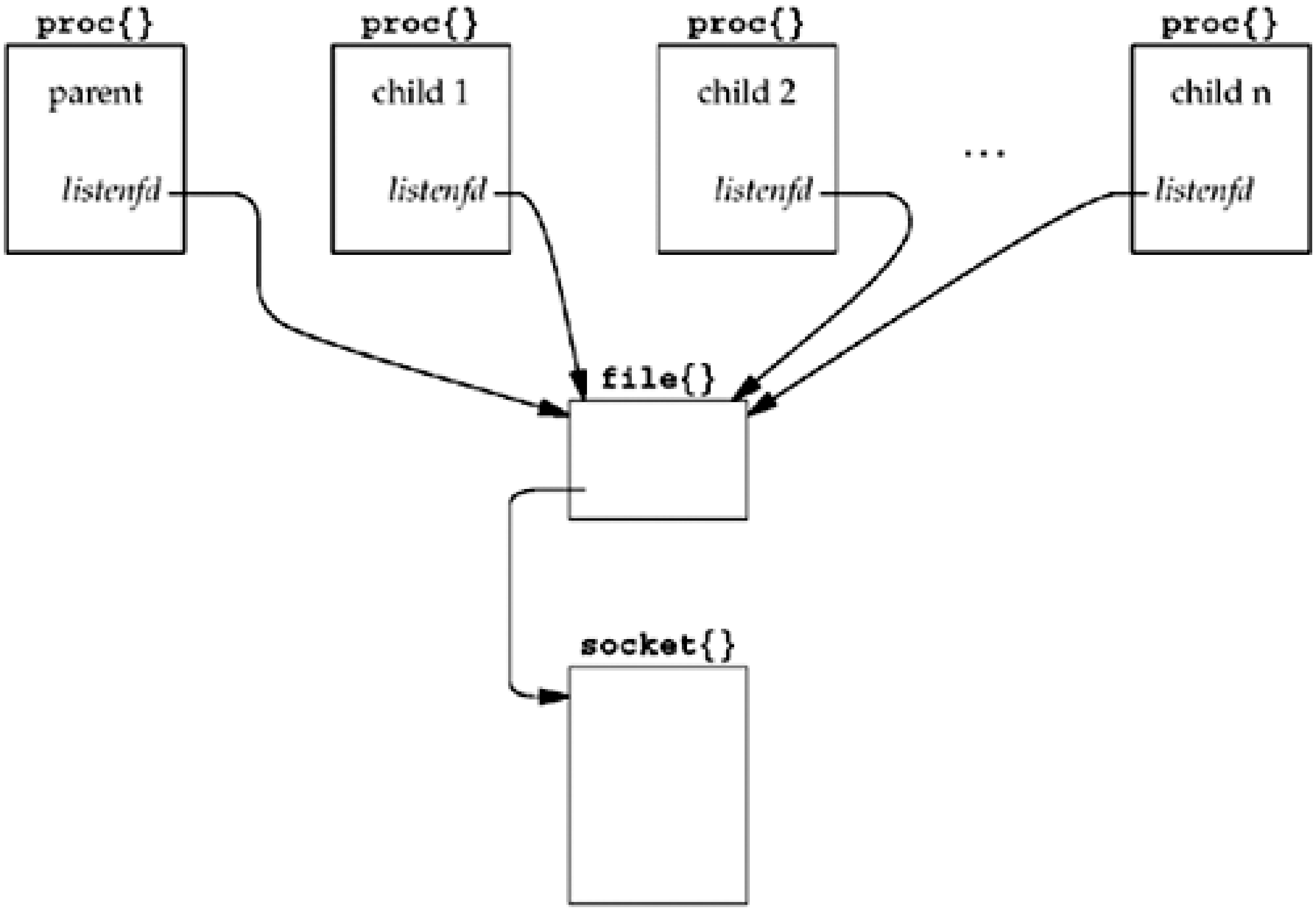
Preforking or Process Pool: No Locking Around Accept

```
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

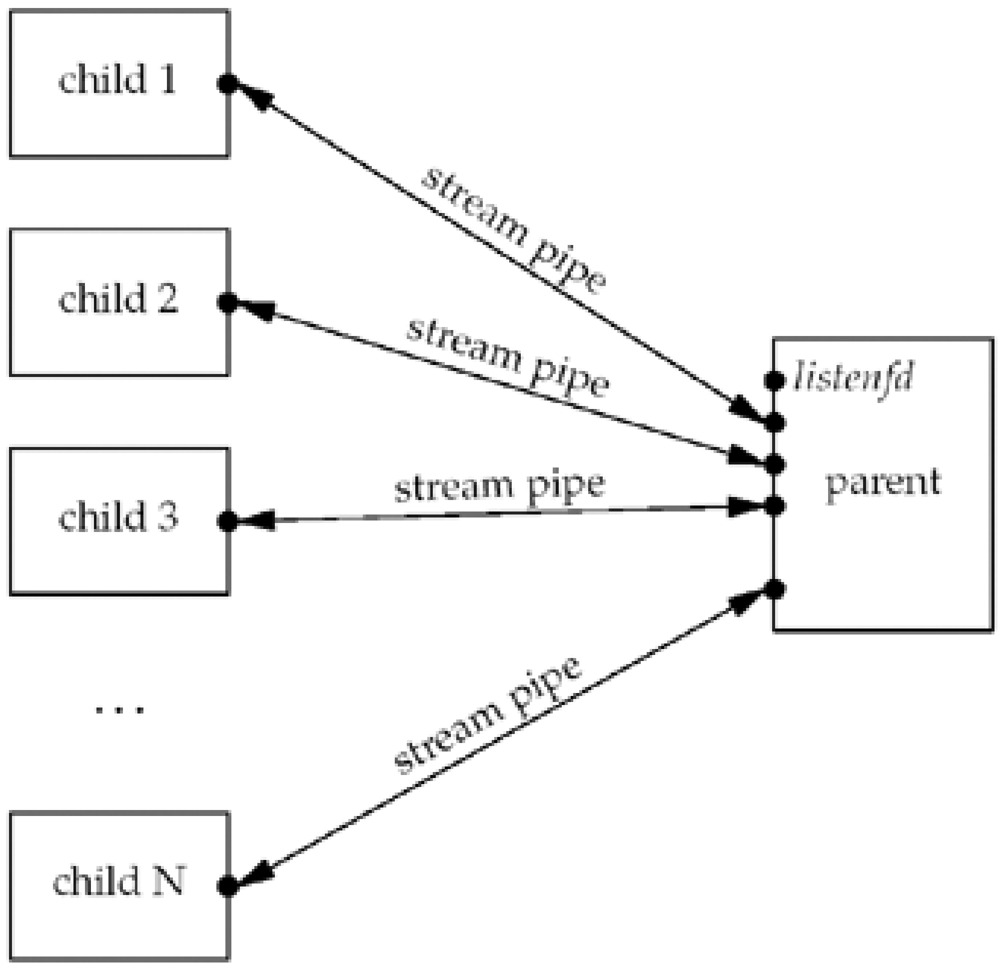
```
static pthread_mutex_t  *mptr;  /* actual mutex will be in shared memory *
void
my_lock_init(char *pathname)
{
        int                 fd;
        pthread_mutexattr_t     mattr;
        fd = Open("/dev/zero", O_RDWR, 0);
        mptr = Mmap(0, sizeof(pthread_mutex_t), PROT_READ | PROT_WRITE,
                                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 */
        }
}
```

innovate achieve lead

BITS Pilani
Pilani Campus

Prethreading

T1: ch 30

Preforked Server Models

- Models:
  - Parent creates pool, child calls accept().
  - Parent creates pool, child calls accept() with a lock around.
    - Child scheduling done by kernel
  - Parent creates pool, parent calls accept(), parent passes connection to child.
    - Child scheduling done by parent
- Advantages
  - Robustness. Even if one child crashes, server keeps running.
  - Simple programming.
- Disadvantages
  - large context switch overheads
  - Large memory footprint per connection. Scalability issue.
  - Optimizations involving sharing information among processes (e.g., caching) harder

Prethread Server Models

- Threads have lower memory foot print and lower context switch overhead.
  - Better scalability
  - They are preferred over processes.
- Instead of creating a new thread every time, a thread pool is created on start up.
- Pthreading Server Models
  - Per-Thread accept()
    - Main thread creates thread pool, and each thread calls accept().
  - main- thread accept()
    - Main thread creates thread pool, calls accept() and pass on the connection to a thread.

Prethreaded Server per-Thread accept()

- Main thread creates *nthreads* and waits for all threads.
- Each thread calls *accept()* with mutex around.

```
int listenfd, nthreads;
socklen_t addrlen;
pthread_mutex_t mlock=PTHREAD_MUTEX_INITIALIZER;
int main(int argc, char **argv)
 {
     int      i;
     void     sig_int(int), thread_make(int);
     listenfd=socket();
     bind(listenfd, );
     nthreads = atoi(argv[argc - 1]);
     for (i = 0;i < nthreads; i++)
         thread_make(i);  /* only main thread returns */
     signal(SIGINT, sig_int);
     for ( ; ; )
         pause();        /* everything done by threads */
 }
```

Prethreaded Server per-Thread accept()

```c
void thread_make(int i)
 {
     void      *thread_main(void *);
     pthread_create(&thread_tid, NULL, &thread_main, (void *) i);
     return; /* main thread returns */
 }

void *thread_main(void *arg)
 {
     int     connfd;
     void    web_child(int);
     socklen_t clilen;
     struct sockaddr *cliaddr;
     cliaddr = malloc(addrlen);
     printf("thread %d starting\n", (int) arg);
     for ( ; ; ) {
         clilen = addrlen;
         pthread_mutex_lock(&mlock);
         connfd = accept(listenfd, cliaddr, &clilen);
         pthread_mutex_unlock(&mlock);
         tptr[(int) arg].thread_count++;
         web_child(connfd);      /* process request */
         close(connfd);
     }
 }
```

Prethreaded Server Main-Thread accept()

- Main thread creates a pool of threads when it starts.
- Main thread calls accept(). Maintains a thread *tptr* array, and *clifd* array*.*
- *clifd* array:
  - A shared array to hold connected fds.
    - Main thread will store.
    - Child threads will take one of these.
    - *iget:* index of the next entry to be fecthed by thread.
    - *iput:* index where next entry will stored.

```
typedef struct {
 pthread_t thread_tid; /* thread ID */
 long     thread_count; /* # connections handled */
} Thread;
Thread *tptr; /* array of Thread structures; calloc'ed */

#define MAXNCLI 32
int     clifd[MAXNCLI], iget, iput;
pthread_mutex_t clifd_mutex;
pthread_cond_t clifd_cond;
```

Prethreaded Server Main-Thread accept()

innovate achieve lead

```
static int nthreads;
pthread_mutex_t clifd_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t clifd_cond = PTHREAD_COND_INITIALIZER;
int main(int argc, char **argv)
  {
     int      i, listenfd, connfd;
     void     sig_int(int), thread_make(int);
     socklen_t addrlen, clilen;
     struct sockaddr *cliaddr;
        listenfd=sock();
        bind();
        listen();
     nthreads = atoi(argv[argc - 1]);
     tptr = calloc(nthreads, sizeof(Thread));
     iget = iput = 0;
     /* create all the threads */
```

Prethreaded Server Main-Thread accept()

```
    /* create all the threads */
    for (i = 0; i < nthreads; i++)
        thread_make(i);             /* only main thread returns */
    signal(SIGINT, sig_int);
    for ( ; ; ) {
        clilen = addrlen;
        connfd = accept(listenfd, cliaddr, &clilen);
        pthread_mutex_lock(&clifd_mutex);
        clifd[iput] = connfd;
        if (++iput == MAXNCLI)
            iput = 0;
        if (iput == iget)
            err_quit("iput = iget = %d", iput);
        pthread_cond_signal(&clifd_cond);
        pthread_mutex_unlock(&clifd_mutex);
    }
}
```

- Condition variable *clifd_cond* is used to communicate the availability of new connection.

Prethreaded Server Main-Thread accept()

```
void * thread_main(void *arg)
 {
     int     connfd;
     void    web_child(int);
     printf("thread %d starting\n", (int) arg);
     for ( ; ; ) {
         pthread_mutex_lock(&clifd_mutex);
         while (iget == iput)
             pthread_cond_wait(&clifd_cond, &clifd_mutex);
         connfd = clifd[iget];   /* connected socket to service */
         if (++iget == MAXNCLI)
             iget = 0;
         pthread_mutex_unlock(&clifd_mutex);
         tptr[(int) arg].thread_count++;
         web_child(connfd);      /* process request */
         Close(connfd);
     }
 }
```

- If iget==iput then there is no new connection. So wait on condition variable.

Comparing Multi Process/Multi Thread Designs

- Maximum 10 simultaneous connections.
- Process pool size:15 / Thread pool size: 15

Model	Process control CPU time (secs)
Iterative	0 (base case)
One fork per client req	20.90
Prefork with child calling accept	1.80
Prefork with child calling accept mutext around	1.75
Prefork with parent passing socket fd to child	2.58
Thread per client req	0.99
Pre threaded with child calling accept	1.93
Prethreaded with main thread calling accept	2.05

- Pre threaded models are better than preforked models.
- Kernel managed connection distribution better perfomance.

Threads vs Processes

- Threads provide better performance than processes.
  - Lower context switch overheads
  - Shared address space simplifies optimizations (e.g., caches)
- But
  - Some extra memory needed to support multiple stacks
  - Need thread-safe programs, synchronization
  - Security: one faulty thread can bring down whole server.
- Apache combines best of both processes and threads using Preforked and prethreaded model.
- IIS on windows platform supports only multi threaded model.

Apache

- Apache is a open source HTTP web server and is built and maintained over at Apache.org
- Apache is comprised of two main building blocks
  - Apache core
  - Apache modules
- Easy to implement and easy to extend its abilities by adding different modules.
- More info at http://www.shoshin.uwaterloo.ca/~oadragoi/cw/CS746G/a1/apache_conceptual_arch.html

Multi-Processing Modules

- Apache 1.3 is a pre-forking server.
  - Easier for UNIX platforms but difficult in Windows platform.
- In Apache 2.0, an abstract layer for Multi-Processing Modules is designed.

Concurrency in Apache

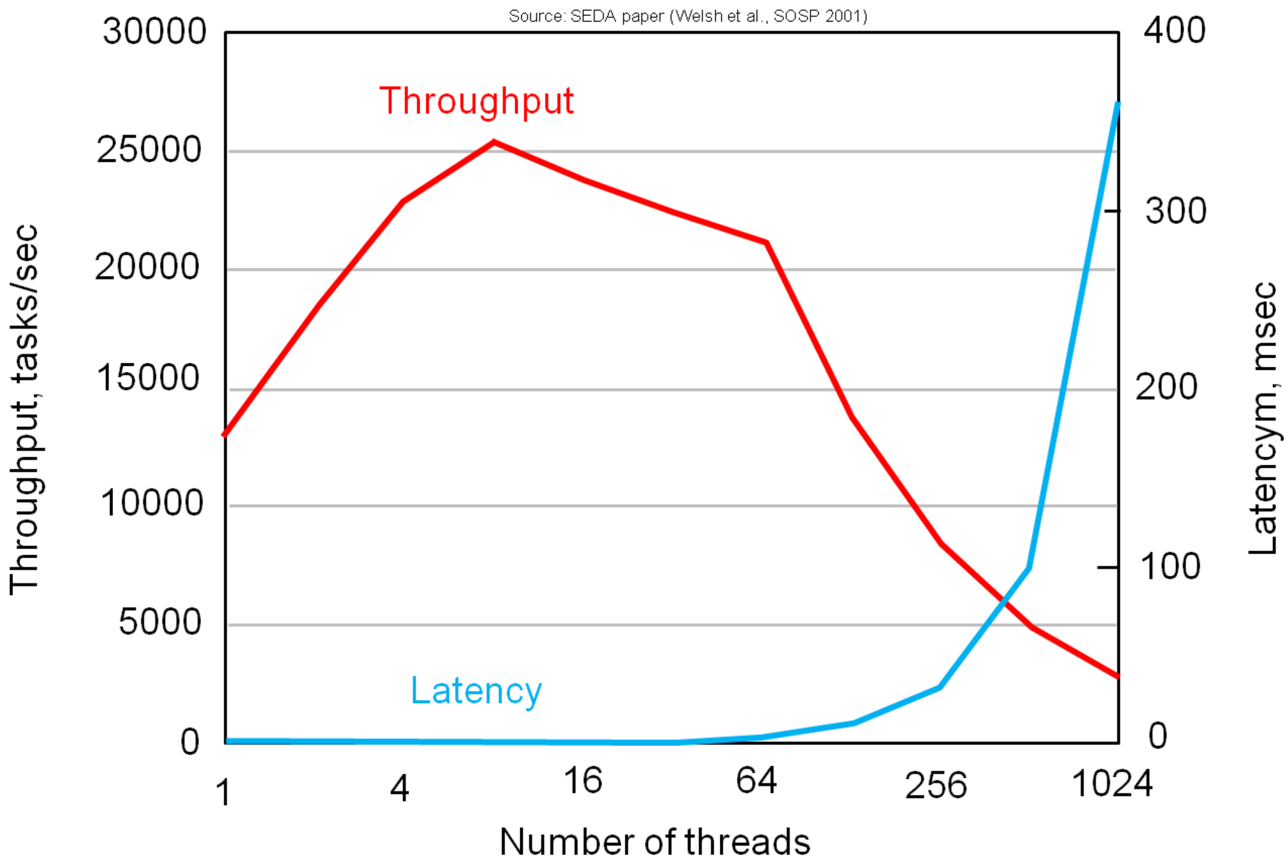
- Following models are supported by Apache
  - prefork
    - Default. Currently the default for Unix and sites that require stability.
  - threaded
    - Suitable for sites that require the benefits brought by threading, particularly reduced memory footprint and improved interthread communications.
  - mpmt_pthread
    - Similar to prefork, but each child process has a specified number of threads. It is possible to specify a minimum and maximum number of idle threads.
  - Dexter
    - Multiprocess, multithreaded MPM that allows you to specify a static number of processes.

innovate achieve lead

BITS Pilani
Pilani Campus

Single Process Server Models

Threads Scalability



- As the number of threads increase in the system, more time is taken by context switching than actually doing productive work.

C10k Problem

- The C10k problem is the problem of providing scalable concurrency to handle a large number of clients at the same time.
- http://www.kegel.com/c10k.html
- Proposes alternatives to thread/process based servers.

Single Process Servers

- We can design a single server process to handle multiple clients employing I/O multiplexing, signal-driven I/O or epoll.
- The server process must take on some of the scheduling tasks that are normally handled by the kernel.
  - Signal-driven I/O
    - a process requests that the kernel send it a signal when input is available or data can be written on a specified file descriptor.
    - When monitoring large numbers of file descriptors, signal-driven I/O performs better than select() and poll().
  - POSIX AIO
    - Linux provides a threads-based implementation of POSIX AIO within glibc. Not widely used.
  - epoll
    - Scalable for large number of fds. Specific to Linux. Please see R1:63.4.

Level-Triggered and Edge-Triggered Notification

- level-triggered interrupts occur whenever the file descriptor is ready for I/O
  - 1000 bytes of data in receive buffer
  - you call recv() and extract 500 bytes
  - select() will continue to indicate the fd is ready because there are still 500 bytes in the buffer
- edge-triggered interrupts occur whenever the file descriptor goes from being not ready to ready
  - 1000 bytes of data in receive buffer . Kernel delivers a signal to owner process.
  - you call recv() and extract 500 bytes
  - Another signal will not be delivered until the receive buffer goes down to zero and then back up to some positive number

Level-Triggered and Edge-Triggered Notification

I/O model	Level-triggered?	Edge-triggered?
select(), poll()	•	
Signal-driven I/O		•
epoll	•	•

- Why epoll() performs better?
  - On each call to select() or poll(), the kernel must check all of the file descriptors specified in the call.
    - But in epoll using epoll_ctl() fd list is created in kernel space.
    - Whenever I/O becomes ready on a fd, kernel adds it to *ready list.* When user calls epoll_wait(), simply return *ready list.*
  - select() passes data to kernel each time it is called.

Number of descriptors monitored (*N*)	*poll()* CPU time (seconds)	*select()* CPU time (seconds)	*epoll* CPU time (seconds)
10	0.61	0.73	0.41
100	2.9	3.0	0.42
1000	35	35	0.53
10000	990	930	0.66

poll()

```
#include <poll.h>
int poll(struct pollfd  fds [], nfds_t  nfds , int  timeout );
/*Returns number of ready file descriptors, 0 on timeout, or -1 on error*/
```

- With select(), we provide three sets, each marked to indicate the file descriptors of interest.
- With poll(), we provide a list of file descriptors, each marked with the set of events of interest.

```
struct pollfd {
    int   fd;        /* File descriptor */
    short events;    /* Requested events bit mask */
    short revents;   /* Returned events bit mask */
};
```

  - The caller initializes *events* to specify the events to be monitored for the file descriptor fd. When poll() returns, *revents* is set to indicate which of those events occurred for this file descriptor.
    - *events* can be 0 if do not want to include that fd.

Events

Table 63-2: Bit-mask values for *events* and *revents* fields of the *pollfd* structure

Bit	Input in *events*?	Returned in *revents*?	Description
POLLIN	•	•	Data other than high-priority data can be read
POLLRDNORM	•	•	Equivalent to POLLIN
POLLRDBAND	•	•	Priority data can be read (unused on Linux)
POLLPRI	•	•	High-priority data can be read
POLLRDHUP	•	•	Shutdown on peer socket
POLLOUT	•	•	Normal data can be written
POLLWRNORM	•	•	Equivalent to POLLOUT
POLLWRBAND	•	•	Priority data can be written
POLLERR		•	An error has occurred
POLLHUP		•	A hangup has occurred
POLLNVAL		•	File descriptor is not open
POLLMSG			Unused on Linux (and unspecified in SUSv3)

- flags of real interest are POLLIN, POLLOUT, POLLPRI, POLLRDHUP, POLLHUP, and POLLERR.

poll() example

```
/* Build the file descriptor list to be supplied to poll(). This list
   is set to contain the file descriptors for the read ends of all of
   the pipes. */
for (j = 0; j < numPipes; j++) {
    pollFd[j].fd = pfds[j][0];
    pollFd[j].events = POLLIN;
}
ready = poll(pollFd, numPipes, -1);         /* Nonblocking */
if (ready == -1)
    errExit("poll");
printf("poll() returned: %d\n", ready);
/* Check which pipes have data available for reading */
for (j = 0; j < numPipes; j++)
    if (pollFd[j].revents & POLLIN)
        printf("Readable: %d %3d\n", j, pollFd[j].fd);
exit(EXIT_SUCCESS);
}
```

Differences between select() and poll()

- When fds are sparsely present, poll() gives better performance than select().
- select() is widely supported.
- Mapping between select() and poll():

```
/*To implement select(), a set of macros is used to convert the information
returned by the kernel poll routines into the corresponding event types returned
by select():*/
#define POLLIN_SET  (POLLRDNORM | POLLRDBAND | POLLIN | POLLHUP | POLLERR)
                                    /* Ready for reading */
#define POLLOUT_SET (POLLWRBAND | POLLWRNORM | POLLOUT | POLLERR)
                                    /* Ready for writing */
#define POLLEX_SET  (POLLPRI)
```

select() & poll()

- CPU time required by select() and poll() increases with the number of file descriptors being monitored.
- The poor scaling performance of select() and poll() stems from :
  - a program makes repeated calls to monitor the same set of file descriptors; however, the kernel doesn’t remember the list of file descriptors to be monitored between successive calls.
- Signal-driven I/O and epoll
  - allow the kernel to record a persistent list of file descriptors
  - scale according to the number of I/O events that occur, rather than according to the number of file descriptors being monitored

Signal-Driven I/O Model

- To use signal-driven I/O with a socket (SIGIO) requires the process to perform the following three steps:
  - A signal handler must be established for the SIGIO signal.
  - The socket owner must be set, normally with the F_SETOWN command of fcntl.
  - Signal-driven I/O must be enabled for the socket, normally with the F_SETFL command of fcntl to turn on the O_ASYNC flag.

Signal-Driven I/O Model

```c
/* Establish handler for "I/O possible" signal */
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART;
sa.sa_handler = sigioHandler;
if (sigaction(SIGIO, &sa, NULL) == -1)
    errExit("sigaction");
/* Set owner process that is to receive "I/O possible" signal */
if (fcntl(STDIN_FILENO, F_SETOWN, getpid()) == -1)
    errExit("fcntl(F_SETOWN)");
/* Enable "I/O possible" signaling and make I/O nonblocking
   for file descriptor */
flags = fcntl(STDIN_FILENO, F_GETFL);
if (fcntl(STDIN_FILENO, F_SETFL, flags | O_ASYNC | O_NONBLOCK) == -1)
    errExit("fcntl(F_SETFL)");
```

```c
static void
sigioHandler(int sig)
{
gotSigio = 1;
}
```

```c
if (gotSigio) { /* Is input available? */
            /* Read all available input until error (probably EAGAIN)
               hash (#) character is read */
            while (read(STDIN_FILENO, &ch, 1) > 0 && !done) {
                printf("cnt=%d; read %c\n", cnt, ch);
                done = ch == '#';
            }
            gotSigio = 0;
        }
    }
```

Signal driven I/O for large no of FDs

- Two more steps:
- Employ fcntl() operation, F_SETSIG , to specify a *realtime signal* that should be delivered instead of SIGIO when I/O is possible on a file descriptor.
  - Realtime signals allow queuing of signals
- Specify the SA_SIGINFO flag when using *sigaction()* to establish the handler for the realtime signal employed in the previous step.
  - si_signo: the number of the signal that caused the invocation of the handler.
  - This value is the same as the first argument to the signal handler.
  - si_fd: the file descriptor for which the I/O event occurred.
  - si_code: a code indicating the type of event that occurred.
  - si_band: a bit mask containing the same bits as are returned in the revents fieldby the poll() system call.

siginfo structure

```
typedef struct {
    int     si_signo;           /* Signal number */
    int     si_code;            /* Signal code */
    int     si_trapno;          /* Trap number for hardware-generated signal
                                   (unused on most architectures) */
    union sigval si_value;      /* Accompanying data from sigqueue() */
    pid_t   si_pid;             /* Process ID of sending process */
    uid_t   si_uid;             /* Real user ID of sender */
    int     si_errno;           /* Error number (generally unused) */
    void   *si_addr;            /* Address that generated signal
                                   (hardware-generated signals only) */
    int     si_overrun;         /* Overrun count (Linux 2.6, POSIX timers) */
    int     si_timerid;         /* (Kernel-internal) Timer ID
                                   (Linux 2.6, POSIX timers) */
    long    si_band;            /* Band event (SIGPOLL/SIGIO) */
    int     si_fd;              /* File descriptor (SIGPOLL/SIGIO) */
    int     si_status;          /* Exit status or signal (SIGCHLD) */
    clock_t si_utime;           /* User CPU time (SIGCHLD) */
    clock_t si_stime;           /* System CPU time (SIGCHLD) */
} siginfo_t;
```

Events

Table 63-7: *si_code* and *si_band* values in the *siginfo_t* structure for "I/O possible" events

si_code	*si_band* mask value	Description
POLL_IN	POLLIN | POLLRDNORM	Input available; end-of-file condition
POLL_OUT	POLLOUT | POLLWRNORM | POLLWRBAND	Output possible
POLL_MSG	POLLIN | POLLRDNORM | POLLMSG	Input message available (unused)
POLL_ERR	POLLERR	I/O error
POLL_PRI	POLLPRI | POLLRDNORM	High-priority input available
POLL_HUP	POLLHUP | POLLERR	Hangup occurred

Server with Signal Driven I/O

```
struct sigaction sa, sa1;
memset (&sa, '\0', sizeof (sa));
memset (&sa, '\0', sizeof (sa1));
sigemptyset (&sa.sa_mask);
sa.sa_flags = SA_SIGINFO;
sa.sa_sigaction = &sigioListenHandler;//for accepting new conn
sigaction (SIGIO, &sa, NULL);
sigaction (SIGRTMIN + 1, &sa, NULL);

sigemptyset (&sa1.sa_mask);
sa1.sa_flags = SA_SIGINFO;
sa1.sa_sigaction = &sigioConnHandler; //for reading data
sigaction (SIGRTMIN + 2, &sa1, NULL);
```

- Two Realtime signals are used. One on listenfd and other on connfds.
  - Unlike standard signals, real-time signals have no predefined meanings.
  - They are queued. They should be defined as SIGRTMIN+n because SIGRTMIN may vary across OSs.

Server with Signal Driven I/O

```
104 listenfd = socket (AF_INET, SOCK_STREAM, 0);
105 bzero (&servaddr, sizeof (servaddr));
106 servaddr.sin_family = AF_INET;
107 servaddr.sin_addr.s_addr = htonl (INADDR_ANY);
108 servaddr.sin_port = htons (atoi (argv[1]));
109 bind (listenfd, (struct sockaddr *) &servaddr, sizeof (servaddr));
110 listen (listenfd, LISTENQ);
111 
112 fcntl (listenfd, F_SETOWN, getpid ());
113 int flags = fcntl (listenfd, F_GETFL);  /* Get current flags */
114 fcntl (listenfd, F_SETFL, flags | O_ASYNC | O_NONBLOCK);//set signal driven IO
115 fcntl (listenfd, F_SETSIG, SIGRTMIN + 1);//replace SIGIO with realtime signal
```

- Line 114: Set listening socket to receive a signal on IO availability.
- Line 115: Replace default signal SIGIO with realtime signal SIGRTMIN+1.

```c
int listenfd; //global var so that signal handlers can access them.
int connfd;
static void
sigioListenHandler (int sig, siginfo_t * si, void *ucontext)
{
printf ("no:%d, for fd:%d,  event band:%ld\n", si->si_signo,
   (int) si->si_fd, (long) si->si_band);
fflush (stdout);
if (si->si_code==POLL_IN)
{
   int n = accept (listenfd, NULL, 0);
   if (n > 0)
connfd = n;
   fcntl (connfd, F_SETOWN, getpid ());
   int flags = fcntl (connfd, F_GETFL);  /* Get current flags */
   fcntl (connfd, F_SETFL, flags | O_ASYNC | O_NONBLOCK);
   fcntl (connfd, F_SETSIG, SIGRTMIN + 2);
}
if (sig == SIGIO)
printf ("Real time signalQ overflow");

}
```

- This handler is for listenfd. It receives siginfo_t when a signal is delivered.
- si_code carries the event that has occurred.
    - Accept connection and set new socket to receive SIGRTMIN+2 signal.

```
44 static void
45 sigioConnHandler (int sig, siginfo_t * si, void *ucontext)
46 {
47 printf ("no:%d, for fd:%d,  , event code:%d,  event band:%ld\n",
48   si->si_signo, (int) si->si_fd, (int) si->si_code,
49   (long) si->si_band);
50 fflush (stdout);
51 if (si->si_code == POLL_IN)
52 {                //input available
53   int n = read (si->si_fd, buf, MAXLINE);
54   if (n == 0)
55 {
56   close (si->si_fd);
57   printf ("Socket %d closed\n", si->si_fd);
58 }
59   else if (n > 0)
60 {
61   buf[n] = '\0';
62   printf ("Data from connfd %d: %s %d\n", connfd, buf, n);
63   write (si->si_fd, "OK", 2);
64 }
```

- At line 51, if the event is POLL_IN, read data from the socket and write back the data to the socket.
- If EOF is received, close the socket.

epoll

- The central data structure of the epoll API is an epoll instance.
- It serves two purposes:
  - recording a list of file descriptors that this process has declared an interest in monitoring—the interest list; and
  - maintaining a list of file descriptors that are ready for I/O—the ready list.
- The epoll API consists of three system calls:
  - The epoll_create() system call creates an epoll instance and returns a file descriptor.
  - The epoll_ctl() system call manipulates the interest list associated with an epoll. Add/del/modify a fd.
  - The epoll_wait() system call returns items from the ready list associated with an epoll instance.

epoll

```c
#include <sys/epoll.h>
int epoll_create(int  size );
/*Returns file descriptor on success, or -1 on error*/
```

```c
#include <sys/epoll.h>
int epoll_ctl(int  epfd , int  op , int  fd , struct epoll_event * ev );
//Returns 0 on success, or -1 on error
```

```c
struct epoll_event {
    uint32_t     events;         /* epoll events (bit mask) */
    epoll_data_t data;           /* User data */
};

typedef union epoll_data {
    void        *ptr;            /* Pointer to user-defined data */
    int          fd;             /* File descriptor */
    uint32_t     u32;            /* 32-bit integer */
    uint64_t     u64;            /* 64-bit integer */
} epoll_data_t;
```

epoll_ctl()

- EPOLL_CTL_ADD
  - Add the file descriptor fd to the interest list for epfd.
- EPOLL_CTL_MOD
  - Modify the events setting for the file descriptor fd, using the information
- EPOLL_CTL_DEL
  - Remove the file descriptor fd from the interest list for epfd.

```c
int epfd;
struct epoll_event ev;
epfd = epoll_create(5);
if (epfd == -1)
    errExit("epoll_create");
ev.data.fd = fd;
ev.events = EPOLLIN;
if (epoll_ctl(epfd, EPOLL_CTL_ADD, fd, ev) == -1)
    errExit("epoll_ctl");
```

epoll_wait()

```
#include <sys/epoll.h>
int epoll_wait(int  epfd , struct epoll_event * evlist , int  maxevents , int  timeout );
//Returns number of ready file descriptors, 0 on timeout, or -1 on error
```

milli secs
-1=blocking, 0=non-blockin
>0 timeout

- The *epoll_wait()* system call returns list of ready file descriptors of epoll instance *epfd.*
- Ready file descriptors is returned in the array of epoll_event structures pointed to by *evlist*.
    - Allocated by caller, *maxevents* is the no of structures in *evlist*.
    - Each structure *evlist* has information about a single ready fd.
    - The *events* subfield returns a mask of the events that have occurred on this fd.
    - The *data* subfield returns whatever value was specified in *ev.data* when we registered interest in this fd using epoll_ctl().
        - data field is the only mechanism for finding out the fd.

epoll Events

Table 63-8: Bit-mask values for the *epoll events* field

Bit	Input to *epoll_ctl()*?	Returned by *epoll_wait()*?	Description
EPOLLIN	•	•	Data other than high-priority data can be read
EPOLLPRI	•	•	High-priority data can be read
EPOLLRDHUP	•	•	Shutdown on peer socket (since Linux 2.6.17)
EPOLLOUT	•	•	Normal data can be written
EPOLLET	•		Employ edge-triggered event notification
EPOLLONESHOT	•		Disable monitoring after event notification
EPOLLERR		•	An error has occurred
EPOLLHUP		•	A hangup has occurred

Server with epoll

```c
55 epfd = epoll_create (20);
56 if (epfd == -1)
57 errExit ("epoll_create");
58 ev.events = EPOLLIN;          /* Only interested in input events */
59 ev.data.fd = listenfd;
60 if (epoll_ctl (epfd, EPOLL_CTL_ADD, listenfd, &ev) == -1)
61 errExit ("epoll_ctl");
62 for (;;)
63 {
64   ready = epoll_wait (epfd, evlist, MAX_EVENTS, -1);
65   if (ready == -1)
66 {
67   if (errno == EINTR)
68     continue;        /* Restart if interrupted by signal */
69   else
70     errExit ("epoll_wait");
71 }
```

- At line no 55, epoll instance is created. At 60, listenfd is added to interest list on event EPOLLIN.
- epoll_wait will block until a fd becomes available.
  - Diff between select() and epoll_wait() is: epoll_wait returns only available fds.

Server with epoll

```
for (j = 0; j < ready; j++)
{
if (evlist[j].events & EPOLLIN)
  {
    if (evlist[j].data.fd == listenfd)
  {
    clilen = sizeof (cliaddr);
    char ip[128];
    memset (ip, '\0', 128);
    int connfd =
      accept (listenfd, (struct sockaddr *) &cliaddr, &clilen);
    ev.events = EPOLLIN;  /* Only interested in input events */
    ev.data.fd = connfd;
    if (epoll_ctl (epfd, EPOLL_CTL_ADD, connfd, &ev) == -1)
      errExit ("epoll_ctl");
  }
```

- Test all returned in evlist array.
- If listenfd is set, accept a new connection. Add new connfd to interest list.
  - Note that we do not need separate client array here. Unless we need them in for statistical purposes.

Server with epoll

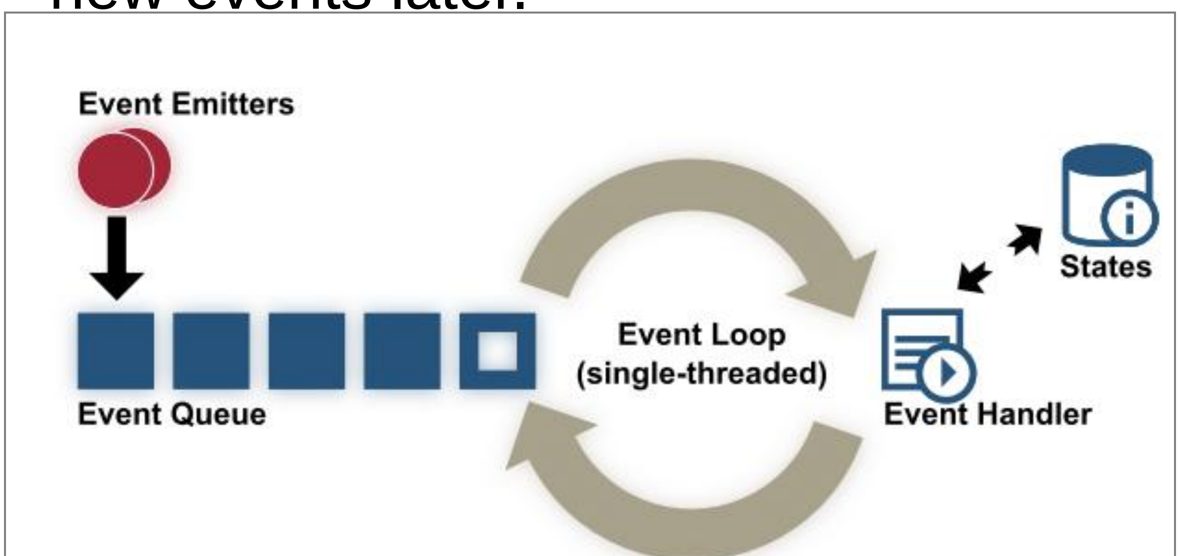
```
    else
    {
      int s = read (evlist[j].data.fd, buf, MAX_BUF);
      buf[s] = '\0';
      if (s == -1)
        errExit ("read");
      if (s == 0)
        {
          close (evlist[j].data.fd);
        }
      if (s > 0)
        write (evlist[j].data.fd, buf, strlen (buf));

    }
    }
```

- If fd is not listenfd, read data, process and send back to the client.
- If EOF is encountered, close the socket.
  - Closing a socket automatically removes it from interest list.

Event Driven Architectures

- A single threaded event loop consumes event after event from the queue and sequentially executes associated event handler code.
- New events are emitted by external sources such as socket or file I/O notifications.
- Event handlers trigger I/O actions that eventually result in new events later.




```
while (1) {
events = getEvents();
for (e in events)
processEvent(e);
}
```

Event Driven Architectures

- Thread is very similar to a scheduler, multiplexing multiple connections to a single flǒw of execution.
- The states of the connections are organized in appropriate data structures— using finite state machines etc.
- Event-driven server architectures is dependent on the availability of asynchronous/non-blocking I/O operations at OS level.

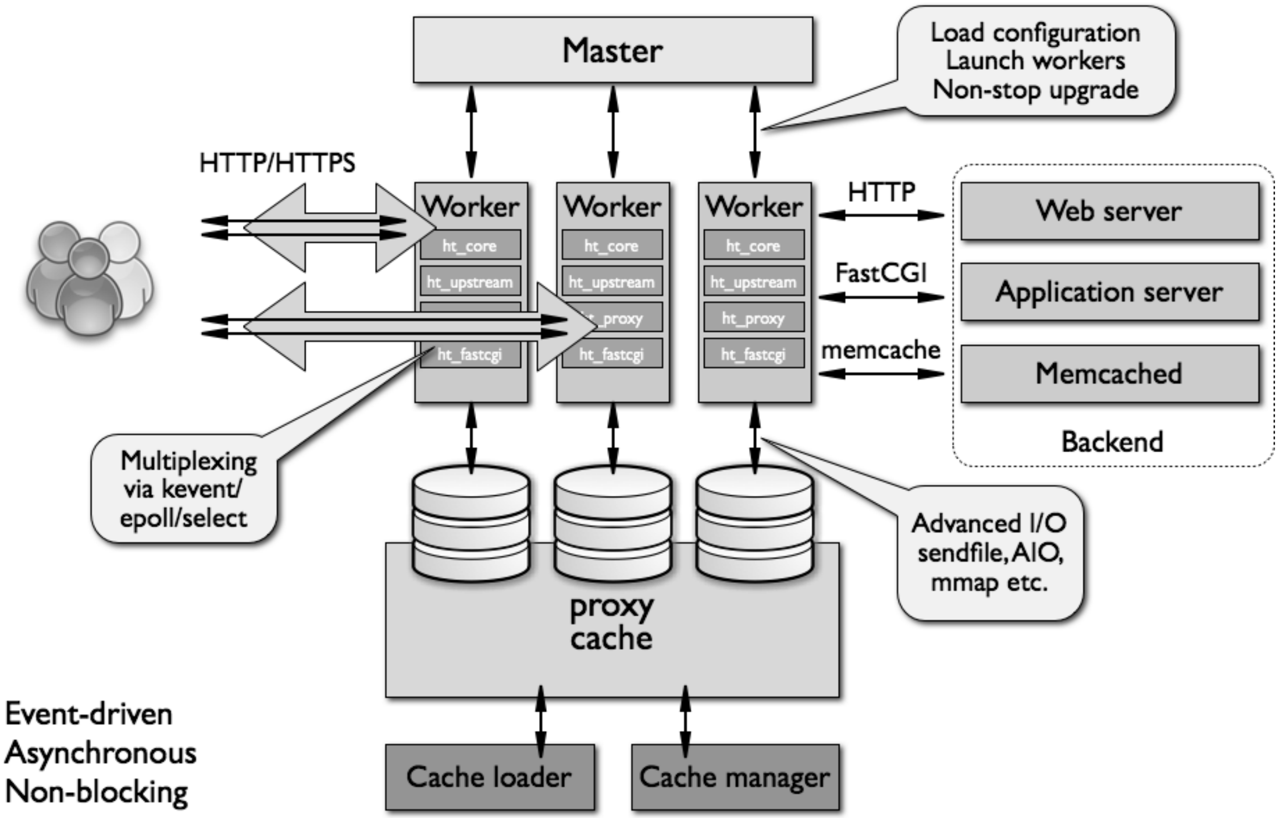
	Blocking	Non-blocking
Synchronous	read/write	read/write using O_NONBLOCK
Asynchronous	I/O multiplexing (select/poll/epoll)	AIO

IO Notification

- IO Multiplexing with Threads
  - Availability of data
  - Copying through a helper thread or process.
  - Notification through a call back function.
- AIO
  - Completion of data.
  - Call back functions which modify the state of the connection and generate events.

Nginx

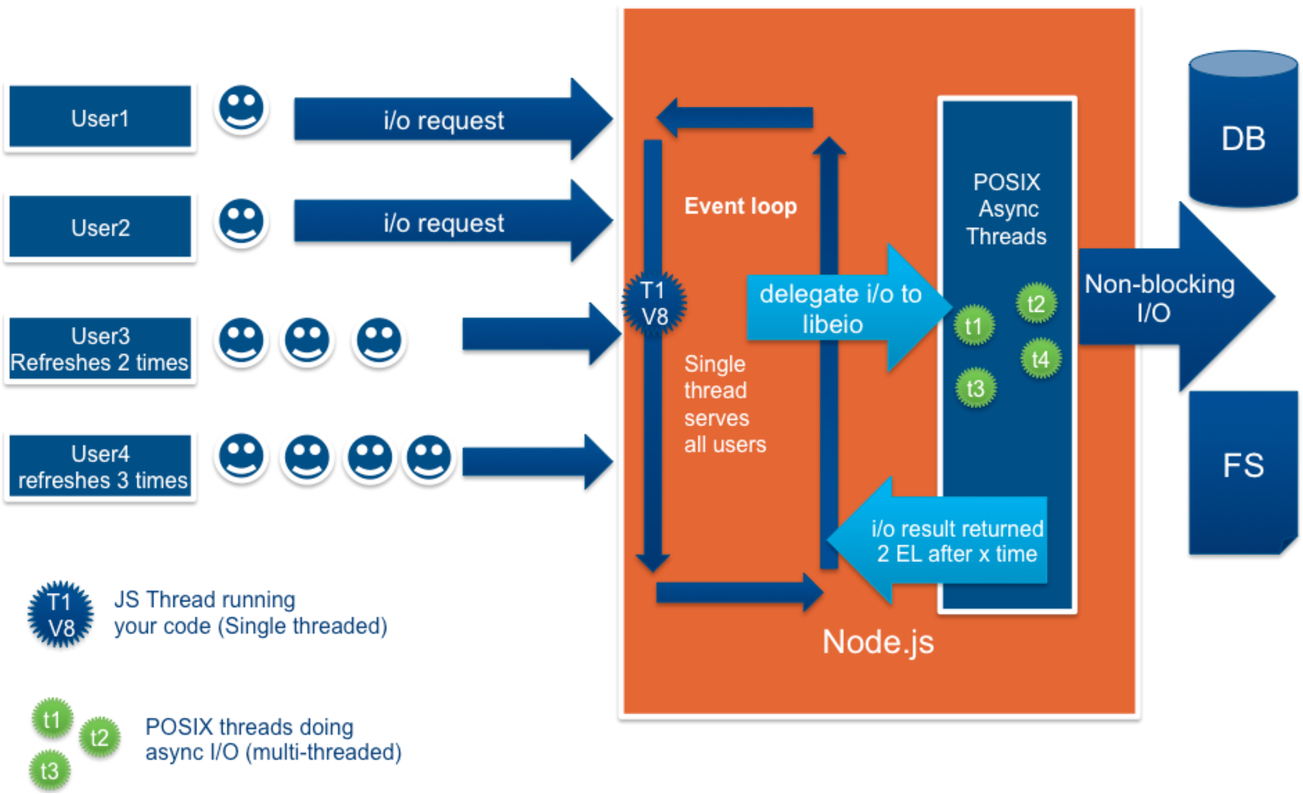
- Multiple worker processes
- Shared listening socket
- Uses Asynchronous callback functions.



- Event-driven
- Asynchronous
- Non-blocking

Node.js

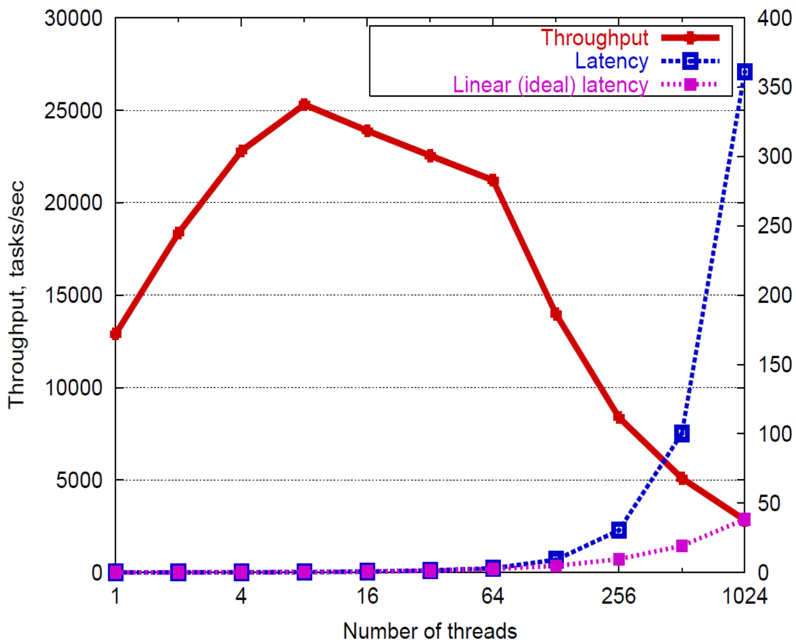
- Single thread
- Asynchronous callback functions



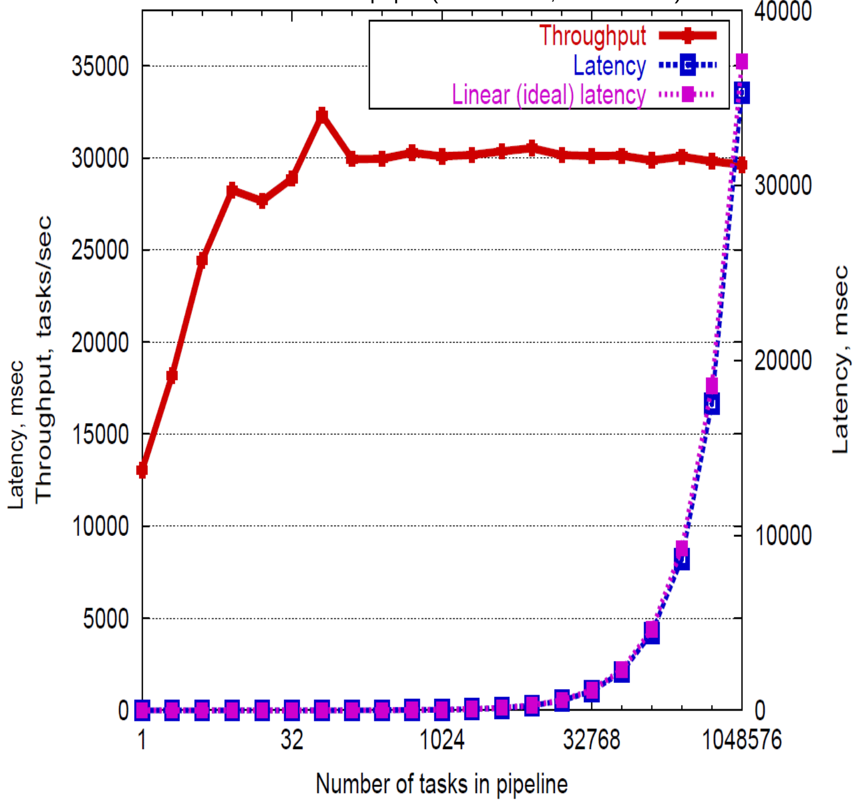
Event Driven Architectures

- Two patterns that involve event demultiplexors are called Reactor and Proactor.
  - The Reactor patterns involve synchronous I/O, whereas the Proactor pattern involves asynchronous I/O.
    - In Reactor, the event demultiplexor waits for events that indicate when a file descriptor or socket is ready for a read or write operation.
    - The demultiplexor passes this event to the appropriate handler, which is responsible for performing the actual read or write.
  - Proactor pattern, the event demultiplexor initiates asynchronous read and write operations.
    - The event demultiplexor waits for events that indicate the completion of the I/O operation, and forwards those events to the appropriate handlers.

Threads vs events



Source: SEDA paper (Welsh et al., SOSP 2001)



- No throughput degradation under load
- Peak throughput is higher

innovate achieve lead

BITS Pilani
Pilani Campus

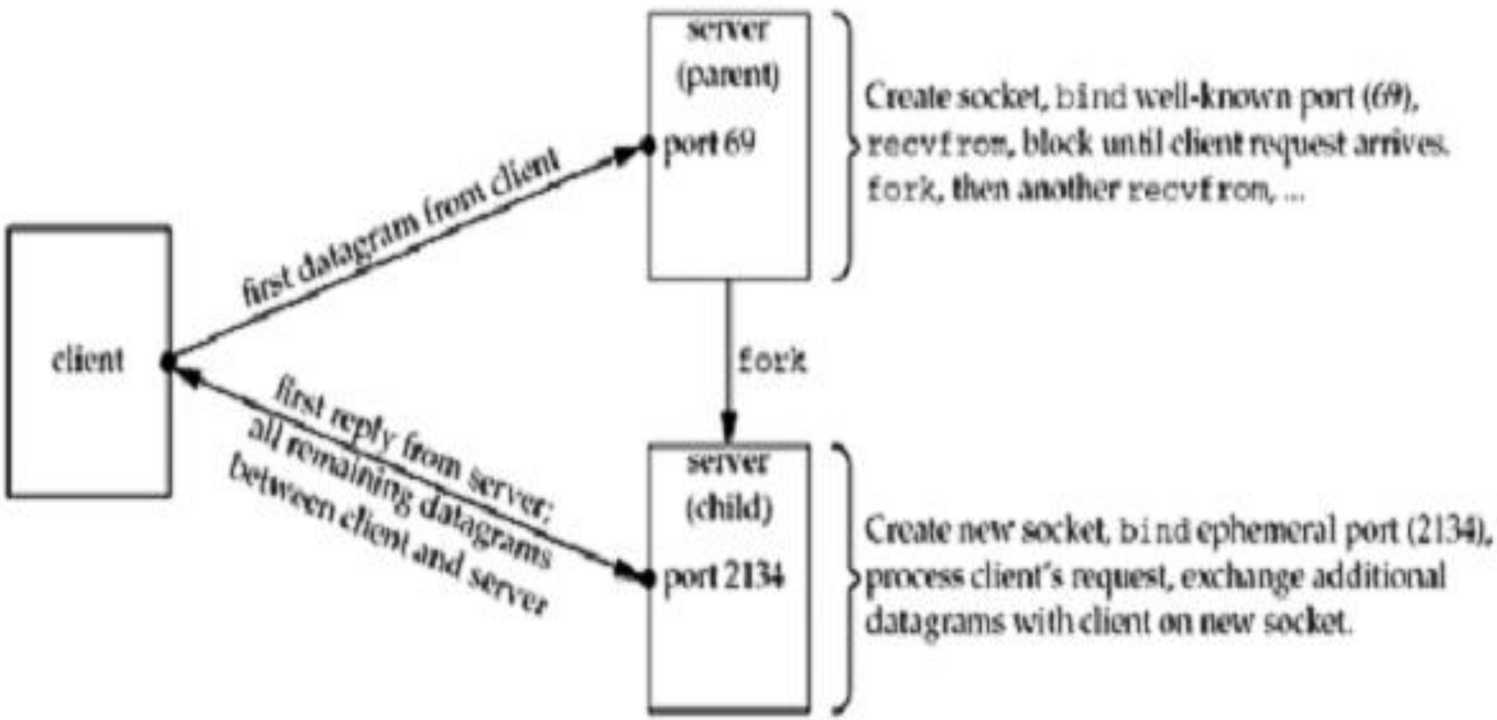
Concurrency UDP Servers

T1: ch 22.7

Concurrent UDP Servers

- Two different types of servers:
- First is a simple UDP server that reads a client request, sends a reply, and is then finished with the client
  - Concurrency: fork a child and let it handle the request
- Second is a UDP server that exchanges multiple datagrams with the client. Extended conersation.
  - Create a new socket for each client, bind an ephemeral port to that socket, and use that socket for all its replies.
  - The client looks at the port number of the server's first reply and send subsequent datagrams to that port.

Concurrency in UDP for Extended Conversations



Acknowledgements

Q&A

Thank You