BITS Pilani
Pilani Campus

Network Programming

K Hari Babu
Department of Computer Science & Information Systems

Outline

Outline

- PreThreading Models
  - Child thread calling accept()
  - Main thread calling accept()
- Case study: Apache
- Processes vs Threads
- Event driven
  - Level-triggered
  - Edge-triggered
- Daemons

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()

```c
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()

```c
    /* 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

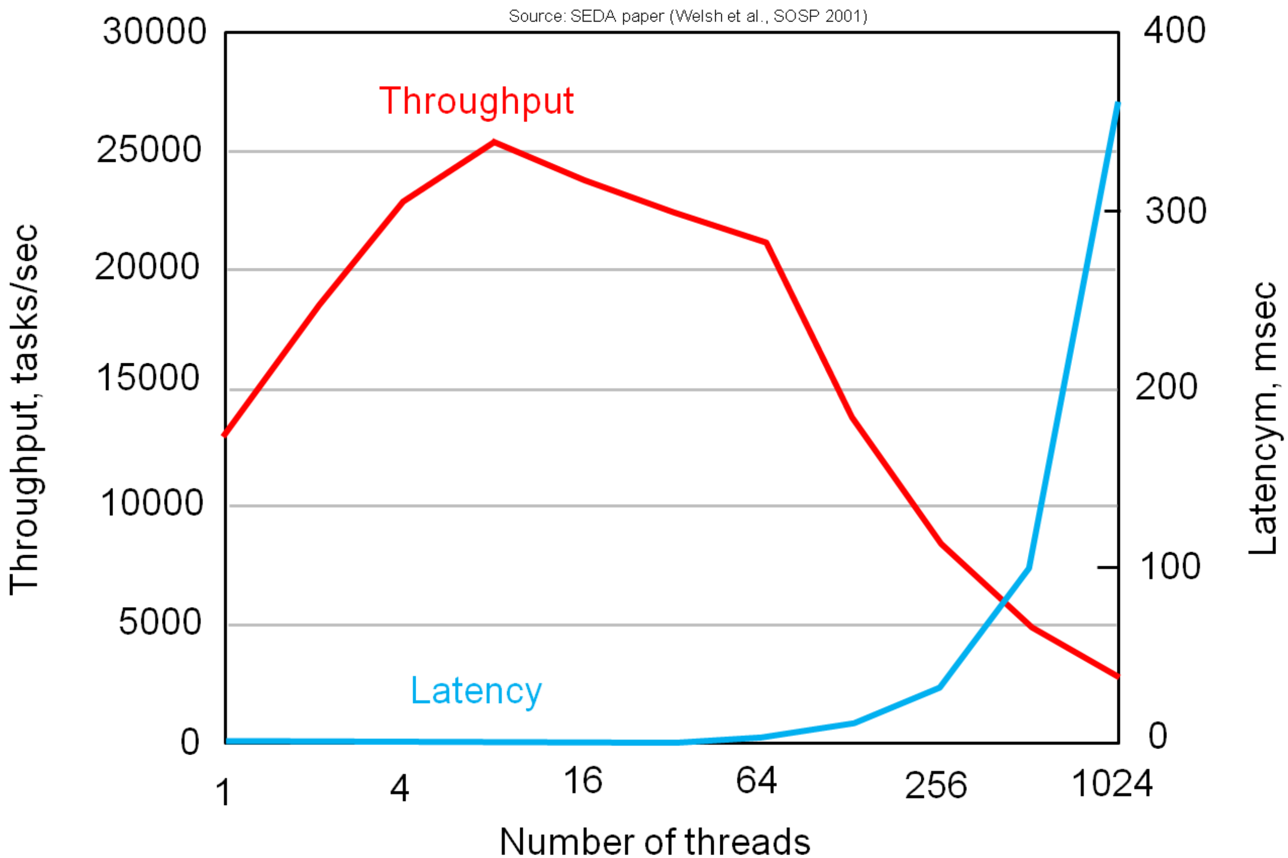
- Apache 1.3 is a pre-forking server.
    - Easier for UNIX platforms but difficult in Windows platform.
- In Apache 2.0, an abstract layer for Multi-Processing Modules is designed.

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.

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 optimising network sockets to handle a large number of clients at the same time.
- http://www.kegel.com/c10k.html

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()

```c
#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.

```c
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

```c
/* 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(0 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

innovate achieve lead

```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

```c
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

```c
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.

```
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.

```c
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

innovate achieve lead

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

```
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.

I/O Multiplexing with Non-blocking I/O

- 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.

Reactor Pattern

- A read() request
  - An event handler declares interest in I/O events that indicate readiness for read on a particular socket
  - The event demultiplexor waits for events
  - An event comes in and wakes-up the demultiplexor, and the demultiplexor calls the appropriate handler
  - The event handler performs the actual read operation, handles the data read, declares renewed interest in I/O events, and returns control to the dispatcher

Reactor Pattern

```c
    select(maxfdp1, &rset, &wset, NULL, NULL);
     if (FD_ISSET(STDIN_FILENO, &rset)) {
        if((n = read(STDIN_FILENO, toiptr, &to[MAXLINE] - toiptr)) < 0) {
            if (errno != EWOULDBLOCK)
                err_sys("read error on stdin");
        }else if (n == 0) {
            fprintf(stderr, "%s: EOF on stdin\n", gf_time());
            stdineof = 1;   /* all done with stdin */
            if (tooptr == toiptr)
                shutdown(sockfd, SHUT_WR);   /* send FIN */
        } else {
        fprintf(stderr, "%s: read %d bytes from stdin\n", gf_time(),
                    n);
            toiptr += n;     /* # just read */
            FD_SET(sockfd, &wset); /* try and write to socket below */
        }
     }
```

Event demultiplexer

Handler

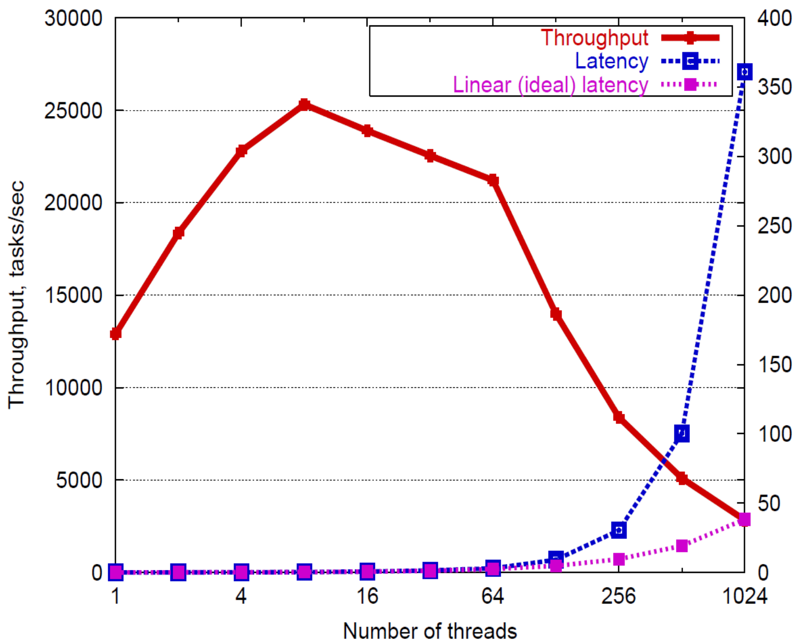
Adding next event

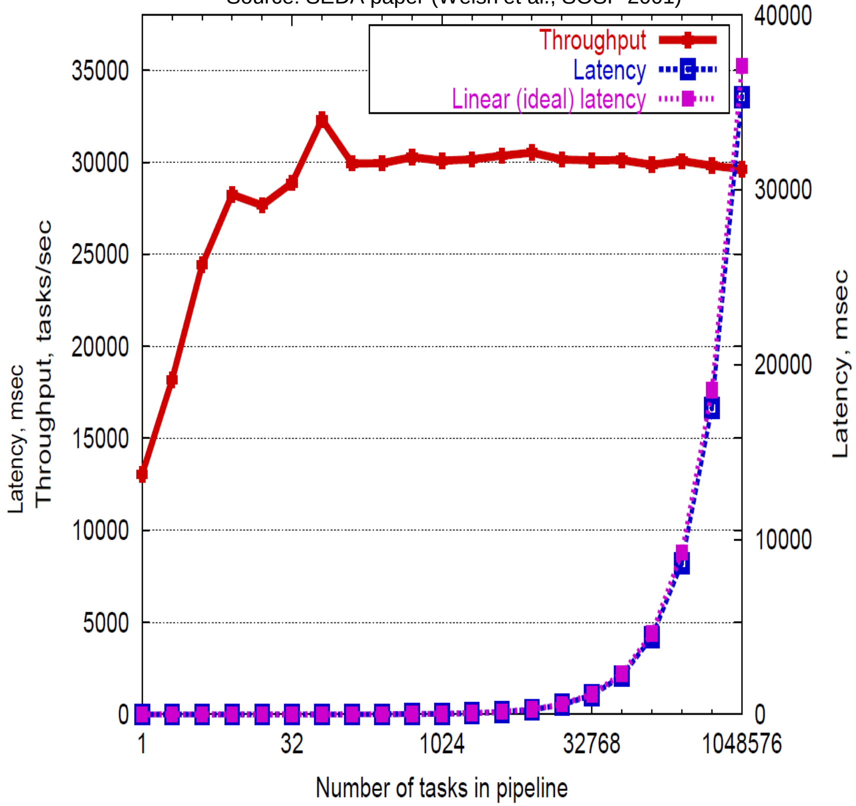
Proactor (true async) Pattern

- A read() request
  - A handler initiates an asynchronous read operation
    - note: the OS must support asynchronous I/O.
    - In this case, the handler does not care about I/O readiness events, but is instead registers interest in receiving completion events.
  - The event demultiplexor waits until the operation is completed
  - While the event demultiplexor waits, the OS executes the read operation in a parallel kernel thread, puts data into a user-defined buffer, and notifies the event demultiplexor that the read is complete
  - The event demultiplexor calls the appropriate handler;
  - The event handler handles the data from user defined buffer, starts a new asynchronous operation, and returns control to the event demultiplexor.

Threads vs events

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





- No throughput degradation under load
- Peak throughput is higher

BITS Pilani
Pilani Campus

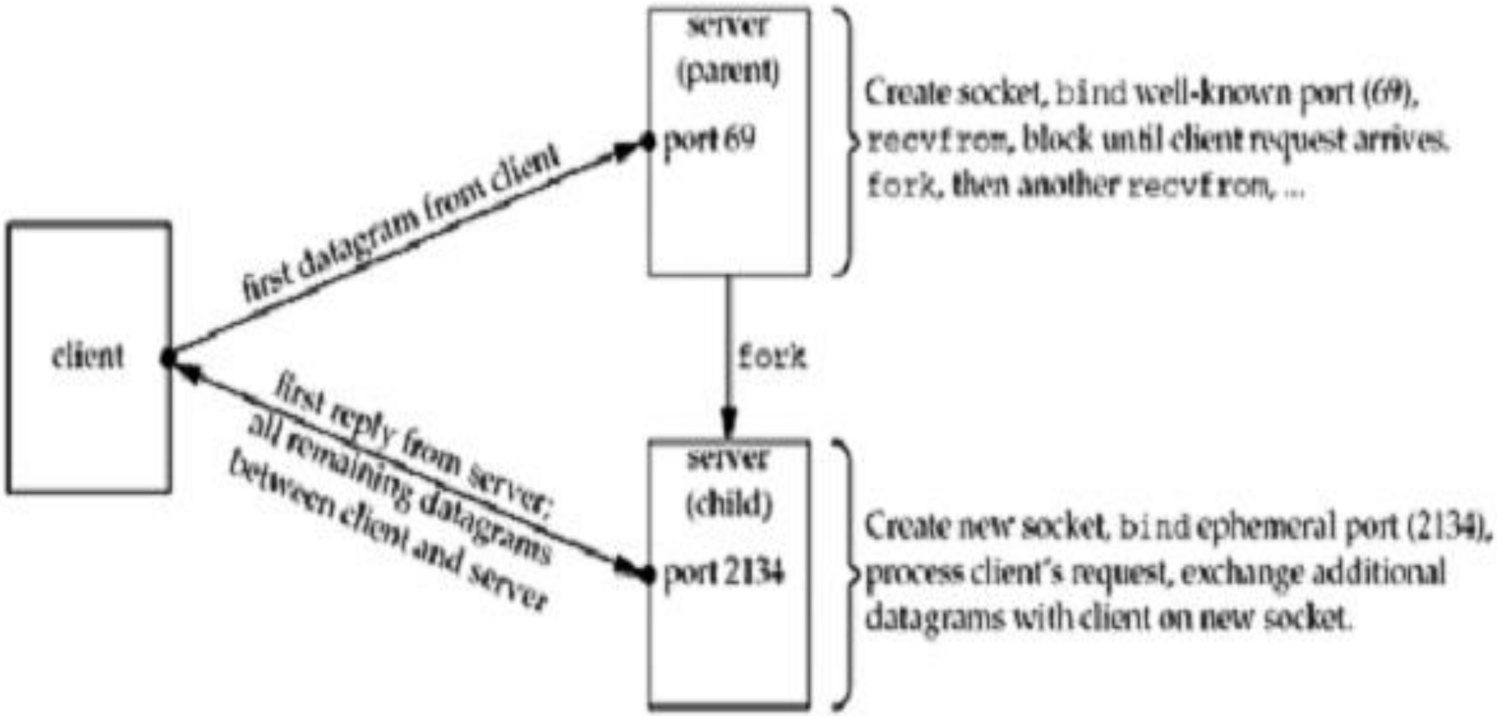
Concurrency UDP Servers

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



innovate achieve lead

BITS Pilani
Pilani Campus

Daemons

R1: ch 37

Daemons

- A daemon is a process with the following characteristics:
  - It is long-lived. Often, a daemon is created at system startup and runs until the system is shut down.
  - It runs in the background and has no controlling terminal. The lack of a controlling terminal ensures that the kernel never automatically generates any job-control or terminal-related signals (such as SIGINT , SIGTSTP , and SIGHUP) for a daemon.
- Examples
  - cron: a daemon that executes commands at a scheduled time.
  - sshd: the secure shell daemon, which permits logins from remote hosts using a secure communications protocol.
  - httpd: the HTTP server daemon (Apache), which serves web pages.
  - inetd: the Internet superserver daemon which listens for incoming network connections on specified TCP/IP ports and launches appropriate server programs to handle these connections.

How to create a Daemon?

```
/* daemons/become_daemon.h*/
#ifndef BECOME_DAEMON_H
#define BECOME_DAEMON_H
/* Bit-mask values for 'flags' argument of becomeDaemon() */
#define BD_NO_CHDIR           01    /* Don't chdir("/") */
#define BD_NO_CLOSE_FILES     02    /* Don't close all open files */
#define BD_NO_REOPEN_STD_FDS  04    /* Don't reopen stdin, stdout, and
                                      stderr to /dev/null */
#define BD_NO_UMASK0         010    /* Don't do a umask(0) */
#define BD_MAX_CLOSE  8192          /* Maximum file descriptors to close if
                                       sysconf(_SC_OPEN_MAX) is indeterminate */
int becomeDaemon(int flags);
#endif
```

How to create a Daemon?

```
int /* Returns 0 on success, -1 on error */
becomeDaemon(int flags)
{
    int maxfd, fd;
    switch (fork()) {    /* Become background process */
    case -1: return -1; /*ensure not a proces sgroup leader*/
    case 0:  break;      /* Child falls through... */
    default: _exit(EXIT_SUCCESS); /* while parent terminates */
    }
    if (setsid() == -1) /* Become leader of new session */
        return -1;
    switch (fork()) {    /* Ensure we are not session leader */
    case -1: return -1;
    case 0:  break;
    default: _exit(EXIT_SUCCESS);
    }
```

How to create a Daemon?

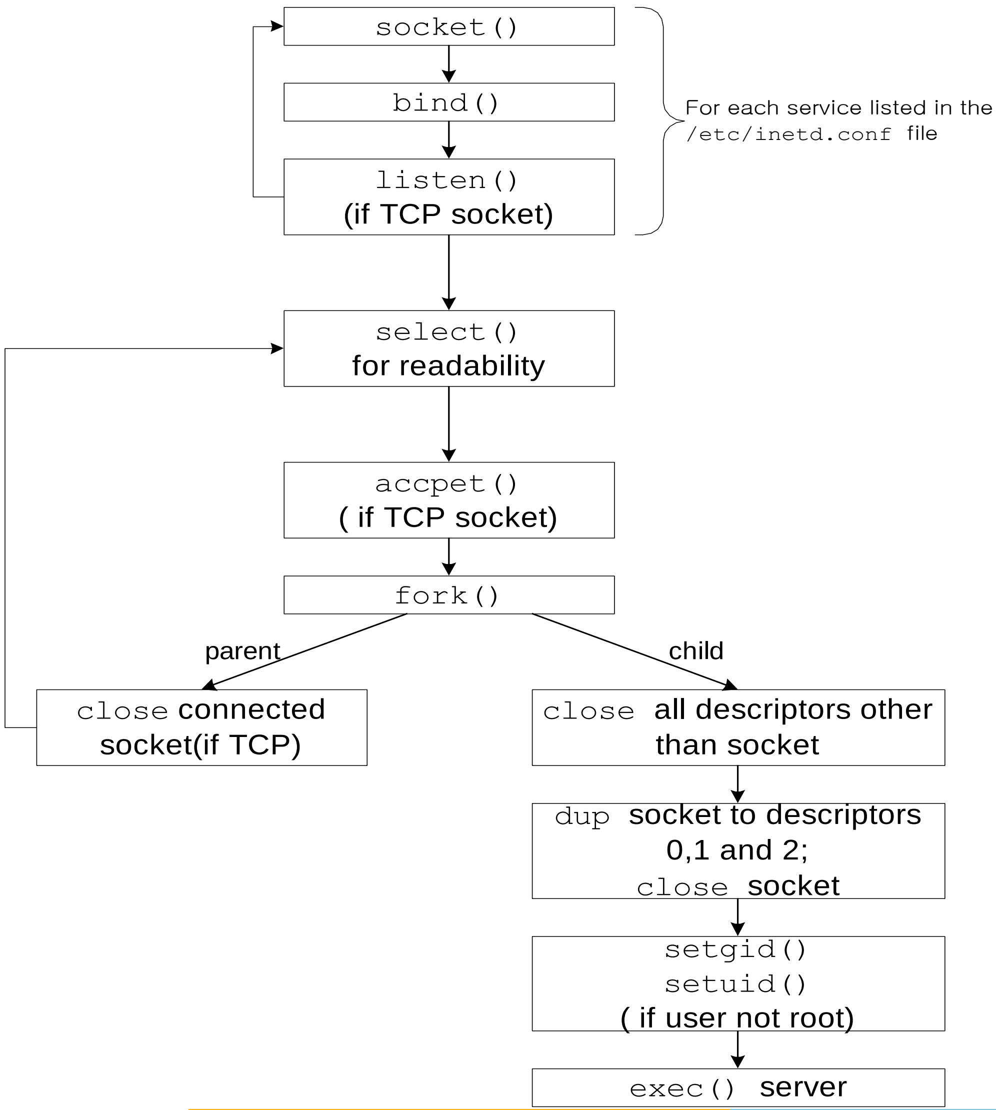
innovate achieve lead

```
        default: _exit(EXIT_SUCCESS);
        }
         if (!(flags & BD_NO_UMASK0))
            umask(0);          /* Clear file mode creation mask */
        if (!(flags & BD_NO_CHDIR))
            chdir("/");  /* Change to root directory */
        if (!(flags & BD_NO_CLOSE_FILES)) { /* Close all open files */
            maxfd = sysconf(_SC_OPEN_MAX);
            if (maxfd == -1)/* Limit is indeterminate... */
                maxfd = BD_MAX_CLOSE; /* so take a guess */
            for (fd = 0; fd < maxfd; fd++)
                close(fd);
        }
        if (!(flags & BD_NO_REOPEN_STD_FDS)) {
            close(STDIN_FILENO);/* Reopen standard fd's to /dev/null */
            fd = open("/dev/null", O_RDWR);
            if (fd != STDIN_FILENO)  /* 'fd' should be 0 */
                return -1;
            if (dup2(STDIN_FILENO, STDOUT_FILENO) != STDOUT_FILENO)
                return -1;
            if (dup2(STDIN_FILENO, STDERR_FILENO) != STDERR_FILENO)
                return -1;
        }
        return 0;
}
```

Inted Super Server

- The inetd daemon is designed to eliminate the need to run large numbers of infrequently used servers. Using inetd provides two main benefits:
  - Instead of running a separate daemon for each service, the inetd daemon monitors a specified set of socket ports and starts other servers as required. Thus, the number of processes running on the system is reduced.
  - The programming of the servers started by inetd is simplified, because inetd performs several of the steps that are commonly required by all network servers on startup.

Inted Super Server



`inetd` service specification

- For each service, `inetd` needs to know:
  - the socket type and transport protocol
  - wait/nowait flag.
  - login name the process should run as.
  - pathname of real server program.
  - command line arguments to server program.
- Servers that are expected to deal with frequent requests are typically *not* run from inetd
  - mail, web, NFS.

/etc/inetd.conf

```
# Syntax for socket-based Internet services:
#  <service_name> <socket_type> <proto> <flags> <user> <server_pathname> <args>
#
# comments start with #
echo      stream  tcp  nowait  root   internal
echo      dgram   udp  wait    root   internal
chargen   stream  tcp  nowait  root   internal
chargen   dgram   udp  wait    root   internal
ftp       stream  tcp  nowait  root   /usr/sbin/ftpd ftpd -l
telnet    stream  tcp  nowait  root   /usr/sbin/telnetd telnetd
finger    stream  tcp  nowait  root   /usr/sbin/fingerd fingerd
# Authentication
auth      stream  tcp  nowait  nobody  /usr/sbin/in.identd in.identd -l -e -o
# TFTP
tftp       dgram   udp   wait    root   /usr/sbin/tftpd tftpd -s /tftpboot
```

wait/nowait

- WAIT specifies that **`inetd`** should not look for new clients for the service until the child (the real server) has terminated.
- TCP servers usually specify **`nowait`** - this means inetd can start multiple copies of the TCP server program - providing concurrency
- Most UDP services run with **`inetd`** told to wait until the child server has died.

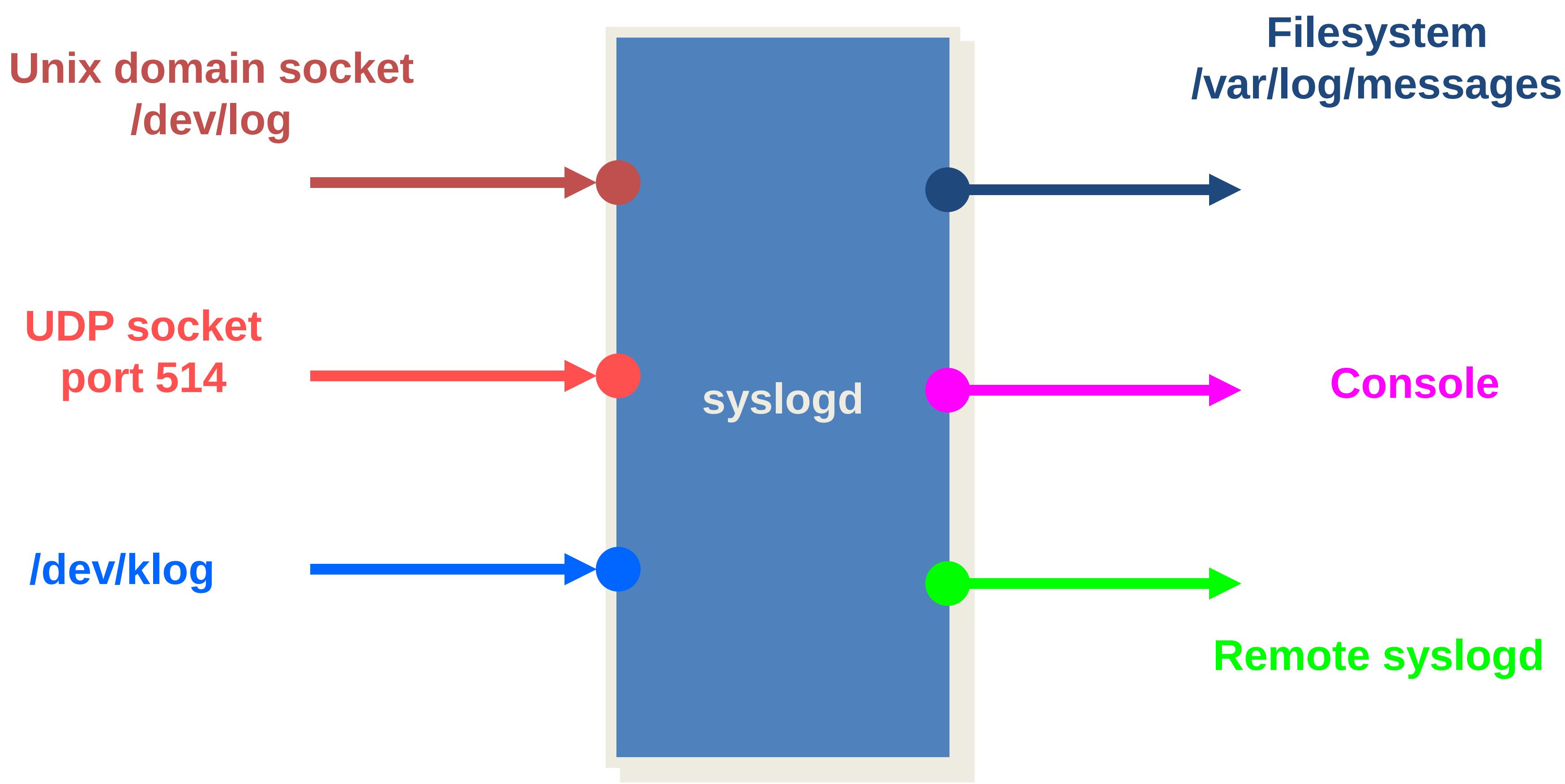
Server Loaded by Inetd

```
/* daytime server*/
int
 main(int argc, char **argv)
 {
	socklen_t len;
	struct sockaddr *cliaddr;
	char	buff[MAXLINE];
	time_t	ticks;
	daemon_inetd(argv[0], 0);
	cliaddr = malloc(sizeof(struct sockaddr_storage));
	len = sizeof(struct sockaddr_storage);
	getpeername(0, cliaddr, &len);
	err_msg("connection from %s", Sock_ntop(cliaddr, len));
	ticks = time(NULL);
	snprintf(buff, sizeof(buff), "%.24s\r\n", ctime(&ticks));
	write(0, buff, strlen(buff));
	close(0);			/* close TCP connection */
	exit(0);
 }
```

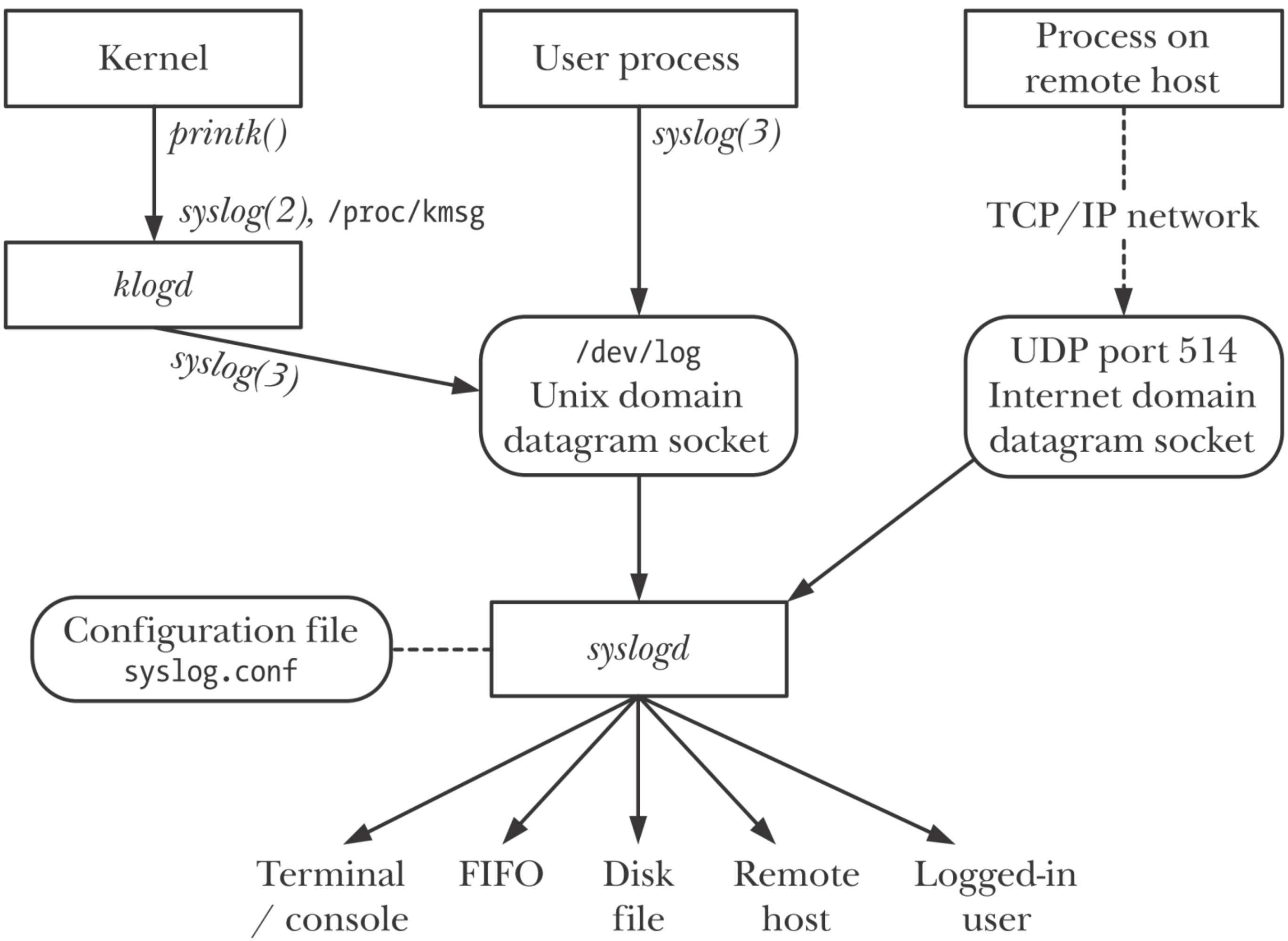
`syslogd` daemon

- Berkeley-derived implementation of `syslogd` perform the following actions upon startup.
    1. The configuration file is read, specifying what to do with each type of log message that the daemon can receive.
    2. A Unix domain socket is created and bound to the pathname `/var/run/log` (`/dev/log` on some system).
    3. A UDP socket is created and bound to port 514
    4. The pathname `/dev/klog` is opened. Any error messages from within the kernel appear as input on this device.
- We could send log messages to the `syslogd` daemon from our daemons by creating a Unix domain datagram socket and sending our messages to the pathname that the daemon has bound, but an easier interface is the `syslog` function.

syslogd


Unix domain socket
/dev/log
UDP socket
port 514
/dev/klog
syslogd
Filesystem
/var/log/messages
Console
Remote syslogd

syslogd


Kernel
printk()
syslog(2), /proc/kmsg
klogd
syslog(3)
User process
syslog(3)
/dev/log
Unix domain
datagram socket
Process on
remote host
TCP/IP network
UDP port 514
Internet domain
datagram socket
Configuration file
syslog.conf
syslogd
Terminal
/ console
FIFO
Disk
file
Remote
host
Logged-in
user

Syslogd

- Each message processed by syslogd has a number of attributes, including a *facility*, which specifies the type of program generating the message, and a *level*, which specifies the severity (priority) of the message.
- The syslogd daemon examines the facility and level of each message, and then passes it along to any of several possible destinations according to the dictates of an associated configuration file, /etc/syslog.conf .

- The syslog API consists of three main functions:
  - The openlog() function establishes default settings that apply to subsequent calls to syslog().
    - The use of openlog() is optional. If it is omitted, a connection to the logging facility is established with default settings on the first call to syslog().
  - The syslog() function logs a message.
  - The closelog() function is called after we have finished logging messages, to disestablish the connection with the log.

```
#include <syslog.h>
void openlog(const char * ident , int  log_options , int  facility );
```

  - The ident argument is a pointer to a string that is included in each message written by syslog();

`syslog` function

```
#include <syslog.h>

void syslog(int priority, const char *message, . . . );
```

- the *priority* argument is a combination of a level and a facility.
- The *message* is like a format string to `printf`, with the addition of a `%m` specification, which is replaced with the error message corresponding to the current value of `errno`.

```
Ex) Syslog(LOG_INFO|LOG_LOCAL2, “rename(%s, %s):
  %m”,file1,file2);
```

Facility

Table 37-1: *facility* values for *openlog()* and the *priority* argument of *syslog()*

Value	Description	SUSv3
LOG_AUTH	Security and authorization messages (e.g., *su*)	•
LOG_AUTHPRIV	Private security and authorization messages	
LOG_CRON	Messages from the *cron* and *at* daemons	•
LOG_DAEMON	Messages from other system daemons	•
LOG_FTP	Messages from the *ftp* daemon (*ftpd*)	
LOG_KERN	Kernel messages (can't be generated from a user process)	•
LOG_LOCAL0	Reserved for local use (also LOG_LOCAL1 to LOG_LOCAL7)	•
LOG_LPR	Messages from the line printer system (*lpr*, *lpd*, *lpc*)	•
LOG_MAIL	Messages from the mail system	•
LOG_NEWS	Messages related to Usenet network news	•
LOG_SYSLOG	Internal messages from the *syslogd* daemon	
LOG_USER	Messages generated by user processes (default)	•
LOG_UUCP	Messages from the UUCP system	•

`syslog` function

level	value	description
LOG_EMERG	0	system is unusable (highest priority)
LOG_ALERT	1	action must be taken immediately
LOG_CRIT	2	critical conditions
LOG_ERR	3	error conditions
LOG_WARNING	4	warning conditions
LOG_NOTICE	5	normal but significant condition (default)
LOG_INFO	6	informational
LOG_DEBUG	7	debug-level message (lowest priority)

level of log message.

- Log message have a level between 0 and 7.

Acknowledgements

Q&A

innovate achieve lead

BITS Pilani
Pilani Campus

Thank You