Network Programming

COSC 1176/1179

Lecture 5
Handling Multiple Connections:
Multiplexing/Forking/Multithreading -II
Lecture Overview

During this lecture, we will cover:

- Assignment issues
- Service Multiple Clients
  - Forking: Pros and Cons of forking servers
  - Pre-forking servers
  - Multiplexing servers
  - Multithreading servers
- Comparisons of various types of servers
Assignment

❖ Have a plan
  ▪ What needs to be done
  ▪ How you will do it
  ▪ What information you need
  ▪ What data and program structures you’ll use

❖ Use the man pages

❖ Experiment with simple code
  ▪ Single sequence (init ... your stuff ... close) client & server
  ▪ Log all actions
  ▪ Check result of all system calls, use errno
Assignment

- **Gotchas**
  - Forget to handle network byte order - htonl() & ntohl()?
  - Forgot to initialise structures, sockets, addresses?

- **Buffers & Memory handling**
  - Take care with start & end of valid data (junk/lost data)
  - Take care with start & end of buffer (SEGVs)

- **Structs**
  - Can you use struct for recv()/send() buffers?
  - Issues - “shape” of structs
    - Data types (size of)
    - Padding

- **Part 4 - using select() - later**
A Forking Server: what we have learnt

- When an application calls `fork()`, an exact duplicate of the calling program is made, and a new child process is started with that copy.
- When the copy begins execution, it starts at the exact place the calling program was, which is the `fork()` call.
Pros of Fork

- It's simple. Creating a new process to handle each client is easy to implement.

- Use processes when using Unix security and user-ids

- Using a process per client keeps any one client from monopolizing the server.

- Other child processes won't be affected if one of the child processes crashes, because the kernel prevents one process from damaging memory in another process.
Cons of Fork()

- The most notable problem with the multi-process approach is difficulty of communicating between cooperating processes.

- Shared memory solutions (for example shmget()) have been made available for multi-process applications, but it isn’t as elegant as threaded approach
  - Shmget() is a system call that allows the allocation of a shared memory segment that can be accessed by multiple processes
  - The way it works is that the parent process creates a shared memory segment upon startup.
    - As each child is created, it inherits the attachment to the shared memory.
    - Access to shared memory must be synchronized - e.g. semaphores.

- With large programs, significant resources can be used because everything must be copied for each child.
Closing Sockets in forked processes

- When the child process is created, everything is copied to the child, including the open file descriptors.

- The kernel keeps a reference count of each descriptor.

- As new children are created, the reference count is incremented for each copy

- Client must close the descriptor of the listening socket used by the parent process

- Parent must close the descriptor of the client socket used by the child process
Closing Sockets in forked processes - example

- Once in the child process, it doesn't need the listening socket any longer.
- So, we close child's reference on that socket.
- However, the socket listensock remains open in the parent process.
- We read characters from the client and echo them to the screen.
- We send the characters back to the client, close the socket, and exit the child process.
- Since the child process has a copy of the client socket, the parent process closes its reference.

Example (code fragments):

```c
listensock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
while (1) {
    newsock = accept(listensock, NULL, NULL);
    if ((pid = fork()) == 0) {
        printf("child process %i created.\n", getpid());
        close(listensock);
        nread = recv(newsock, buffer, 25, 0);
        buffer[nread] = '\0';
        printf("%s\n", buffer);
        send(newsock, buffer, nread, 0);
        close(newsock);
        printf("child process %i finished.\n", getpid());
        exit(0); }
    else
        close(newsock);
}
```
Danger with fork()

If one is not careful, fork() may often create unwanted processes called zombies:

- Normally, you fork() a child and then wait() for its termination.
- Zombies are child processes where the parent process hasn't called wait() or waitpid(). The kernel keeps the exit information for these child processes until the parent process retrieves it by calling waitpid().

```
yallara.cs.rmit.edu.au% ps -l
  F S    UID   PID  PPID   C PRI NI     ADDR     SZ    WCHAN TTY         TIME CMD
0 Z 20064505 27643 26072   0  40 20        ?    196        ? pts/612     0:00 sleep
0 S 20064505 26072 26040   0  40 20        ?    496        ? pts/612     0:00 tcsh
0 O 20064505 28381 26072   0  77 20        ?    222          pts/612     0:00 ps
yallara.cs.rmit.edu.au%
```
Danger with fork()

- Zombie behavior is system (i.e. OS) specific.

- If the parent exits without retrieving the exit information, the child processes usually remains in a zombie state.
  - Zombies consume system resources until they are cleaned up.
  - These are inherited and cleaned-up by init, if the parent terminates.

- What happens if parent hasn't terminated (yet)?
Waitpid()}

```c
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

- Waits for child process to change state (terminate)
- 'pid' specifies the process id of the child process that the parent wishes to wait for.
  - -1 denotes interest in any child
- 'status' will hold the the status of the child when waitpid() returns
- Options can take variety of arguments.

  - If WNOHANG is set then waitpid() will return 0 if the child has not yet terminated.
  - Use option WNOHANG if you want to sit in a loop monitoring a situation, but not blocking on waitpid().
Signal handlers

```c
void sigchld_handler(int signo)
{
    while (waitpid(-1, NULL, WNOHANG) > 0);
}
```

- This signal handler simply calls `waitpid()` for any exited children.
- The handler may be called once when several children have exited.

```c
signal(SIGCHLD, sigchld_handler);
```

- Install the handler before forking

- This helps prevent zombies from occurring
Another signal handler

- Can be handled by trapping the SIGCHLD using sigaction() and calling wait().

```c
void my_handler(int signum)
{
    int status, pid;

    if (-1 != (pid = wait(&status)))
        ...
    else
        // child has finished
        // wait failed
}

{
    struct sigaction sa;

    memset(&sa, 0, sizeof(sa));
    sa.sa_handler = my_handler;
    if (0 != sigaction(SIGCHLD, sa, NULL))
        ...
        // sigaction failed
}
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/wait.h>
#include <signal.h>

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

int main(int argc, char *argv[])
{
    struct sockaddr_in sAddr;
    int listensock;
    int newsock;
    char buffer[25];
    int result;
    int nread;
    int pid;
    int val;
}
Multiprocessing echo server

listensock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);

val = 1;
result = setsockopt(listensock, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
if (result < 0) {
    perror("failure to set listening socket option");
    return 0;
}

sAddr.sin_family = AF_INET;
sAddr.sin_port = htons(1972);
sAddr.sin_addr.s_addr = INADDR_ANY;

result = bind(listensock, (struct sockaddr *) &sAddr, sizeof(sAddr));
if (result < 0) {
    perror("could not bind address to listening socket - already in use?".gateway
    return 0;
}
result = listen(listensock, 5);
if (result < 0) {
    perror("could not listen using listening socket");
    return 0;
}

signal(SIGCHLD, sigchld_handler);
while (1) {
    newsock = accept(listensock, NULL ,NULL);
    if ((pid = fork()) == 0) {
        printf("child process %i created.
", getpid());
        close(listensock);
        nread = recv(newsock, buffer, 25, 0);
        buffer[nread] = '\0';
        printf("%s\n", buffer);
        send(newsock, buffer, nread, 0);
        close(newsock);
        printf("child process %i finished.
", getpid());
        exit(0);
    }
    close(newsock);
}
close(newsock);
Server Models

- On-demand forking
- Process pools
Preforking servers - creating process pools

- For large applications creating a copy of running process and launching it is expensive in terms of time and resources.

- Solution: mitigate startup costs by creating a number of processes into a “process pools” when the application starts. This is called preforking.

[Diagram showing process pool with parent process, server processes, and clients.]
Preforking servers – creating process pools

- With preforking, when a client connects, the process to handle it has already been created. Since the child is already running, there is no process creation delay.
- When a client connects, the kernel chooses one of the children to handle the connection.
- Using this method, accept() is not called in the parent process.
- Unlike the previous example, the listening socket descriptor will not be closed in the child process.
- All the children will be calling accept() on the same listening socket()
preforked echo server -server3.c

Code fragment of preforked Echo server

for (x = 0; x < nchildren; x++) {
    if ((pid = fork()) == 0)
        DoChildProcessing();
    else if (pid < 0)
        fprintf(stderr, "cannot create child processes (errno %i).\n", errno);
}

void DoChildProcessing() {
    while (1) {
        newsock = accept(listensock, NULL ,NULL);
        printf("client connected to child process %i.\n", getpid());
        nread = recv(newsock, buffer, 25, 0);
        buffer[nread] = '\0';
        printf("%s\n", buffer);
        send(newsock, buffer, nread, 0);
        close(newsock);
        printf("client disconnected from child process %i.\n", getpid());
    }
}
Preforking example: apache web server

- The apache web server (version <2.0) uses process pools.

- In apache, process pool size is dynamic - maximum number of children, min and max number of idle children are configurable.

- Specifying the minimum and maximum number of idle children allows the server to handle sudden spikes in usage.

- The parent process continually checks to see how many are idle.

- Terminates extra children or creates new children depending on demand.

- Apache version 2 uses thread pools.
Multithreading

- Threads are lightweight processes that share the same memory space of the parent process.
- Because they share the same memory space, they use fewer resources than a multi-process (forking) application.
- They also enjoy faster context-switch time.
- Multithreaded applications are not as stable as multi-process applications. One bad thread can impact other threads and bring down the entire server program.
Threads and Processes

- One process, one thread
- One process, multiple threads
- Multiple processes, one thread per process
- Multiple processes, multiple threads per process
User-Level Threads (ULT)

- In this level, the kernel is not aware of the existence of threads. All thread management is done by the application by using a thread library.
- Thread switching does not require kernel mode privileges (no mode switch) and scheduling is application specific.
- The kernel is not aware of thread activity but it is still managing process activity.
  - One blocking thread will block the process, blocks all threads!
- Advantages:
  - Thread switching does not involve the kernel.
  - Scheduling can be application specific - you can choose the best algorithm.
  - ULTs can run on any OS - only needs a thread library.
Kernel-Level Threads (KLT)

- All thread management is done by kernel.
- Uses system calls rather than thread library.
- The kernel maintains context information for the process and the threads.
- Switching between threads requires the kernel.
- Scheduling is performed on a thread basis.

**Advantages:**
- The kernel can simultaneously schedule many threads of the same process on many processors.
- Threads can execute (and block) independently of each other.
Hybrid ULT/KLT (solaris OS)

- Thread creation done in the user space
- The programmer may adjust the number of KLTs
- Process includes the user's address space, stack, and process control block
- User-level threads (threads library) invisible to the OS
- each LWP supports one or more ULTs and maps to exactly one KLT
Creating a Thread

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

- The `pthread_create()` function is used to create a new thread, with attributes specified by `attr`, within a process.
- If `attr` is NULL, the default attributes are used.
- On success, the identifier of the newly created thread is stored in the location pointed by the `thread` argument, and a 0 is returned. On error, a non-zero error code is returned.
- The thread is created executing `start_routine` with `arg` as its sole argument.
- Default thread creation:
  ```c
  #include <pthread.h>
  
  void *start_func(void *arg)
  {
  ... }
  
  pthread_t tid;
  pthread_create(&tid, NULL, start_func, arg);
  ```
Thread Arguments

When `start_routine` is called the value `arg` specified in the call to `pthread_create()` is passed as a parameter.

`start_routine` can have only 1 parameter, and it can't be larger than the size of a pointer (void *).

```c
result = pthread_create(&thread_id, NULL, thread_proc, (void *) newsock);

void* thread_proc(void *arg)
{
    int sock, nread;
    char buffer[25];

    printf("child thread %i with pid %i created.\n", pthread_self(), getpid());
    sock = (int) arg;
    nread = recv(sock, buffer, 25, 0);
    buffer[nread] = '\0';
    printf("%s\n", buffer);
    send(sock, buffer, nread, 0);
    close(sock);
    printf("child thread %i with pid %i finished.\n", pthread_self(), getpid());
}
```
Waiting for Thread Termination

```c
int pthread_join(pthread_t thread, void **status);
```

- It keeps the parent thread from continuing until the child thread completes. thread must be a member of the current process and it cannot be a detached thread.

- This function provides a simple mechanism allowing an application to wait for a thread to terminate. After the thread terminates, the application may then choose to clean up the resources that were used by the thread. For instance, after `pthread_join()` returns, any application-provided stack storage could be reclaimed.

- If successful, `pthread_join()` returns 0. Otherwise, an error number is returned to indicate the error. If two or more threads wait for the same thread to complete, all will suspend processing until the thread has terminated.

- Default value of `status` is NULL
- If `status` is not NULL, it points to a location that is set to the exit status of the terminated thread when `pthread_join()` returns successfully.

- An example:
  ```c
  #include <pthread.h>
  pthread_t tid;
  int ret;

  /* waiting to join thread "tid" without status */
  ret = pthread_join(tid, NULL);
  ```
Detaching Thread

int pthread_detach(pthread_t thread);

- The `pthread_detach()` function is used to indicate to the implementation that storage for the specified thread can be reclaimed when the thread terminates.

- If `tid` has not terminated, `pthread_detach()` does not cause it to terminate.

- `pthread_detach()` returns a zero when it completes successfully. Any other returned value indicates that an error occurred.

- A simple example of calling this function to detach a thread is given by:

```c
#include <pthread.h>

pthread_t tid;
int ret;

ret = pthread_detach(tid);
```
Pthread_detach() and zombies

- A zombie is a process (or, thread in this case) that has terminated and is waiting for its parent to check its return value.

- The system will keep the zombies around until the return value is checked, so they take up system resources.

- Pthread_detach() can keep zombies from occurring.

- By using pthread_detach() function we can tell the system that storage for the specified thread can be reclaimed when the thread terminates.
Thread ID - pthread_self()

- Each thread has a unique ID, a thread can find out its ID by calling pthread_self().
- Thread IDs are of type pthread_t which is usually an unsigned int. When debugging, it's often useful to do something like this:

```c
printf("Thread %u:\n",pthread_self());
```
Thread Termination

To stop a child thread, call

```c
void pthread_exit(void *value_ptr);
```

or just return from the thread function.

**DO NOT** `exit()` from a child thread

What will happen if you do?
#include <stdio.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
#include <sched.h>

void* thread_proc(void *arg);

int main(int argc, char *argv[])
{
    struct sockaddr_in sAddr;
    int listensock;
    int newsock;
    int result;
    pthread_t thread_id;
    int val;

    listensock = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    See server4.c
Multithread server example - server4.c

```c
val = 1;
result = setsockopt(listensock, SOL_SOCKET, SO_REUSEADDR, &val, sizeof(val));
if (result < 0) {
    perror("server4");
    return 0;
}
sAddr.sin_family = AF_INET;
sAddr.sin_port = htons(1972);
sAddr.sin_addr.s_addr = INADDR_ANY;

result = bind(listensock, (struct sockaddr *) &sAddr, sizeof(sAddr));
if (result < 0) {
    perror("exserver4");
    return 0;
}
result = listen(listensock, 5);
if (result < 0) {
    perror("exserver4");
    return 0;
}
```
while (1) {
    newsock = accept(listensock, NULL, NULL);
    result = pthread_create(&thread_id, NULL, thread_proc, (void *) newsock);
    if (result != 0) {
        printf("Could not create thread.\n");
        return 0;
    }
    pthread_detach(thread_id);
}

void* thread_proc(void *arg)
{
    int sock;
    char buffer[25];
    int nread;
    printf("child thread %i with pid %i created.\n", pthread_self(), getpid());
    sock = (int) arg;
    nread = recv(sock, buffer, 25, 0);
    buffer[nread] = '\0';
    printf("%s\n", buffer);
    send(sock, buffer, nread, 0);
    close(sock);
    printf("child thread %i with pid %i finished.\n", pthread_self(), getpid());
}
Compiling Threaded Program

```
gcc server4.c -o server4 -lnsl -lsocket -lpthread
```
Prethreaded servers - creating thread pools

- Thread pools operate in a very similar manner to process pools
- The strategy is to create a certain number of threads when the application initializes
- We have a pool of threads to handle incoming client connections.
- Resize our thread pool depending on the load requirements
- We avoid the costs associated with waiting to create a thread when the request is made.
Prethreading example - creating pools

- Create a number of threads `nchildren` in the pool
- The parent process uses a loop to spawn the requested number of threads passing the descriptor of the listening socket
- *Calls `pthread_join()` to keep it from returning before any of its child threads.*

```c
for (x = 0; x < nchildren; x++) {
    result = pthread_create(&thread_id, NULL, thread_proc, (void *) listensock);
    if (result != 0) {
        printf("Could not create thread.\n");
        return 0;
    }
}
...

pthread_join (thread_id, NULL);
...
```
Prethreading - each thread calling accept()

```c
void* thread_proc(void *arg)
{
    int listensock, sock;
    char buffer[25];
    int nread;
    listensock = (int) arg;

    while (1) {
        sock = accept(listensock, NULL, NULL);
        printf("client connected to child thread %i with pid %i.\n",
               pthread_self(), getpid());
        nread = recv(sock, buffer, 25, 0);
        buffer[nread] = '\0';
        printf("%s\n", buffer);
        send(sock, buffer, nread, 0);
        close(sock);
        printf("client disconnected from child thread %i with pid %i.\n",
               pthread_self(), getpid());
    }
}
```

- Each thread calls `accept()` on the same listening socket and waits for a client connection.
- When a connection is made, the operating system chooses one of the threads.
- The chosen thread receives data from the client and echoes it back.
- When the connection is closed the thread calls `accept()` again to wait for another client.

See server5.c
Hybrid Preforking and Prethreading

Hybrid strategy gives speed of prethreading combined with the stability of preforking

Multiprocessing server: stable, but slower context switching
Multithreading server: not stable, but faster context switching.
Example: apache version >= 2.0
Multiplexing

- Multiplexing is a way to handling multiple clients is a single server process.
- The application allows clients to connect to the server and adds them to the watch list.
- The watch list is just an array of socket descriptors.
- The operating system tells the application which clients need to be served or if a new client has established a connection.
Uses of Multiplexed I/O

- Client handling multiple descriptors
  - usually user input and a socket
  - can be more than one socket (rare)
- For a TCP server to handle more than one socket
  - a listening socket and one or more connected sockets
- For a server handling both TCP and UDP
- For a server which provides more than one service
  - listens on one socket per service
  - listens on one socket per client
Server Process

- Server uses `select()` to listen to new connections, check for client disconnects, and read events on existing connections.
- If a read event occurs on the server’s listening socket, then a new connection is initiated and the server calls `accept` to get the new socket descriptor.
- The new descriptor is then added to the server’s watch list.
Why multiplexed I/O? Locking Up

Locking = program becomes unresponsive

- Fixed sequences are generally used as part of a typical client server application interaction: eg for SMTP
  - First you do a HELO, then you wait for a response,
  - Then you do a MAIL FROM:, and wait for a response
  - Now we do a RCPT TO:, etc

- For your time-of-day client

- Fixed sequence programming fails if a step in the sequence causes a block!

- If we are to use this style of programming we need a way of getting around the issue of sitting and waiting forever in a blocked condition.
Easiest way is (for a client) is to use a timeout (calls still block but they don’t block forever)
  - You can use socket option but not portable:
  - Select’s timeout is just as easy and has the advantage of better code portability.

Other methods:
  - Signals and Events
  - Don’t use fixed sequence programming...
Locking Up - select()

• Select() allows our program to wait until there is something of interest for us to process.

• Rather than doing things in a strict sequence we can just do those things which need doing.
select()

- int select( int maxfd,
              fd_set *readset,
              fd_set *writeset,
              fd_set *exceptset,
              const struct timeval *timeout);

- `maxfd` is one higher than the highest file descriptor value to be considered

- three sets of file descriptors (can be NULL)
  - readset : file descriptors you're interested in reading from
  - writeset : file descriptors you're interested in writing to
  - exceptset : file descriptors you're checking for special conditions

- `timeout` structure specifies the longest you want to wait
Using select()

- First, set *readset*, sometimes *writeset*, and *maxfd*
  - almost never need to use *exceptset*
- Then call `select()`
- Then on return:
  - readset contains only those descriptors which were originally in readset and are now readable
  - writeset contains only those descriptors which were originally in writeset and are now writeable
- Return value:
  - the number of descriptors ready, if any;
  - 0, if it timed out;
  - -1 if it fails.
Manipulating fd_set Sets

Four macros provided to manipulate these sets:

- `void FD_SET(int fd, fd_set &fdset);`
  - flags a descriptor to be watched
- `void FD_CLR(int fd, fd_set &fdset);`
  - resets the flag set
- `int FD_ISSET(int fd, fd_set &fdset);`
- `void FD_ZERO(fd_set &fdset);`

Practical use:

- Use `FD_ZERO` to create an empty set, then add descriptors using `FD_SET`
- Call `select()`
- Check which descriptors are set using `FD_ISSET`
Sample Code

```c
fd_set readset;
struct timeval tv;

FD_ZERO(&readset);
FD_SET(socketfd,&readset);

tv.tv_sec = delay;
tv.tv_usec = 0;

if ( select(fd+1,&readset,NULL,NULL,&tv) > 0 ) {
    /* do a read */
}
```
Using select()’s Timeout

- `struct timeval` {
  - long tv_sec;
  - long tv_usec;
};

- To block indefinitely, set `timeout=NULL`
- To achieve a timeout, create & set a `timeval` structure and set `timeout` as a pointer to it
- Can poll, set a `timeval` structure to 0 secs and 0 usecs, but why?
Next Time

- Discuss Signals and Sockets