B336 Advanced Internet Computing

Implementing a Web Server

Learning Objective

- Understand the basic HTTP request-response handling mechanisms of a web server.

- Know the basic requirements of using Perl to implement the HTTP request-response handling mechanisms of a web server.

Lecture Outline

- Sockets
- An example minimal code for components of a web server

A Serial Web Server

- Here's your solution to Ass 1 part II …well, almost!

```
#!/usr/bin/perl
# file: web_serial.pl
# Figure 14.2: The baseline server handles requests serially

use strict;
use IO::Socket;
use Web;

my $port = shift || 8080;
my $socket = IO::Socket::INET->new( LocalPort => $port,
                                    Listen    => SOMAXCONN,
                                    Reuse     => 1 )
                          or die "Can't create listen socket: $!";
while (my $c = $socket->accept) {
  handle_connection($c);
  close $c;
}
close $socket;
```

- The `Web.pm` package used by your web server script:

```
package Web;
# file: Web.pm
# Figure 15.1: Core Web Server Routines

# utility routines for a minimal web server.
# handle_connection() and docroot() are only exported functions

use strict;
use vars '@ISA','@EXPORT';
require Exporter;

@ISA = 'Exporter';
@EXPORT = qw(handle_connection docroot);

my $DOCUMENT_ROOT = '/home/www/htdocs';
my $CRLF = "\015\012";
```

(continued next page)

```
sub handle_connection {
  my $c = shift;   # socket
  my ($fh,$type,$length,$url,$method);
  local $/ = "$CRLF$CRLF";   # set end-of-line character
  my $request = <$c>;        # read the request header

  return invalid_request($c)
    unless ($method,$url) = $request =~ m!^(GET|HEAD) (/.*) HTTP/1\.[01]!;
  return not_found($c) unless ($fh,$type,$length) = lookup_file($url);
  return redirect($c,"$url/") if $type eq 'directory';

  # print the header
  print $c "HTTP/1.0 200 OK$CRLF";
  print $c "Content-length: $length$CRLF";
  print $c "Content-type: $type$CRLF";
  print $c $CRLF;

  return unless $method eq 'GET';

  # print the content
  my $buffer;
  while ( read($fh,$buffer,1024) ) {
    print $c $buffer;
  }
  close $fh;
}
```

(continued next page)

```
sub lookup_file {
  my $url = shift;
  my $path = $DOCUMENT_ROOT . $url;                # turn into a path
  $path =~ s/\?.*$//;                              # get rid of query
  $path =~ s/\#.*$//;                              # get rid of fragment
  $path .= 'index.html' if $url=~m!/$!;            # get index.html if path ends in /
  return if $path =~ m!/\.\./!;                    # don't allow relative paths (..)
  return (undef,'directory',undef) if -d $path;    # oops! a directory
  my $type = 'text/plain';                         # default MIME type
  $type = 'text/html'  if $path =~ /\.html?$/i;    # HTML file?
  $type = 'image/gif'  if $path =~ /\.gif$/i;      # GIF?
  $type = 'image/jpeg' if $path =~ /\.jpe?g$/i;    # JPEG?
  return unless my $length = (stat(_))[7];         # file size
  return unless my $fh = IO::File->new($path,"<");   # try to open file
  return ($fh,$type,$length);
}
```

(continued next page)

```
sub redirect {
  my ($c,$url) = @_;
  my $host = $c->sockhost;
  my $port = $c->sockport;
  my $moved_to = "http://$host:$port$url";
  print $c "HTTP/1.0 301 Moved permanently$CRLF";
  print $c "Location: $moved_to$CRLF";
  print $c "Content-type: text/html$CRLF$CRLF";
  print $c <<END;
<HTML>
<HEAD><TITLE>301 Moved</TITLE></HEAD>
<BODY><H1>Moved</H1>
<P>The requested document has moved
<A HREF="$moved_to">here</A>.</P>
</BODY>
</HTML>
END
}
```

(continued next page)

```
sub invalid_request {
  my $c = shift;
  print $c "HTTP/1.0 400 Bad request$CRLF";
  print $c "Content-type: text/html$CRLF$CRLF";
  print $c <<END;
<HTML>
<HEAD><TITLE>400 Bad Request</TITLE></HEAD>
<BODY><H1>Bad Request</H1>
<P>Your browser sent a request that this server
does not support.</P>
</BODY>
</HTML>
END
}
```

(continued next page)

```
sub not_found {
  my $c = shift;
  print $c "HTTP/1.0 404 Document not found$CRLF";
  print $c "Content-type: text/html$CRLF$CRLF";
  print $c <<END;
<HTML>
<HEAD><TITLE>404 Not Found</TITLE></HEAD>
<BODY><H1>Not Found</H1>
 <P>The requested document was not found on this server.</P>
</BODY>
</HTML>
END
}

sub docroot {
  $DOCUMENT_ROOT = shift if @_;
  return $DOCUMENT_ROOT;
}

1;
```

(continued next page)

Sockets

- Sockets is a programming language concept which allows a TCP or UDP connections to be formed between two network programs.
    - They serve as “end-points” of the TCP or UDP connection - something Internet client and server code use to send data to the other side.
    - Programmer establish these connections by setting parameters like port numbers, transport protocol, etc.
    - The socket API handles the details or TCP, UDP, IP, etc - this is the concept of ***layered software***.

- In Perl, the common way of using sockets is to create the `IO::Socket` objects.

Sockets in your web client

- Why didn’t we use sockets in our web client in the last lecture?

- We did!
  - The `LWP::UserAgent` class uses sockets to send requests for us.
  - If you look in the file C:\>PERL\SITE\LIB\PROTOCOL\HTTP.PM, you will see the similar calls to the IO::Socket class methods as in our web server.

More about Sockets

- You can learn the details of how to use socket APIs in the unit **B310 UNIX and Network Programming**.

- In this unit, we are more interested in the HTTP request-response handling mechanism built on top of the

Our Serial Web Server

- In our example **serial** web server, we only deal with one connection from one client at a time.
- This is sufficient to demonstrate the concept of request-response handling on our server.
- Extending this to a more realistic multiprocessing or multithreading web server which can serve multiple request at once (such as what the Apache web server does) is beyond this unit.

Our Serial Web Server

- For illustration purpose, we also only restrict ourselves to only processing “GET” and “HEAD” requests.

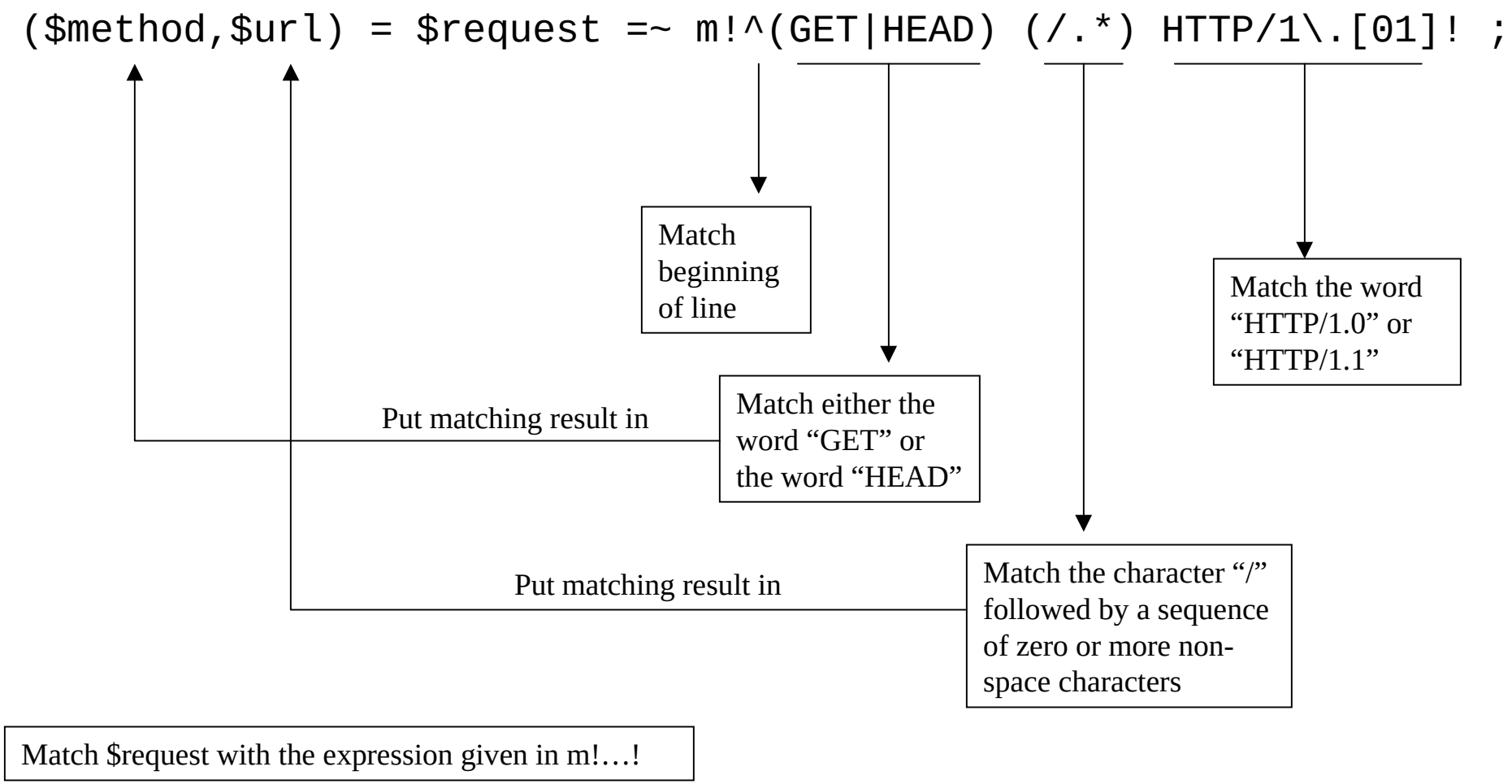
Outline of serial_web.pl

```
<COMMENTS>

<LOAD PACKAGES>

<Get the PORT number from the first argument to the script, or set
to 8080 if the script has no arguments>

<Create an Internet TCP socket for the given PORT number -
“Listen” is the maximum queue size for the socket, SOMAXCONN is
the a system variable - Setting “Reuse” to 1 means the same socket
can be used for more than one connection>

<LOOP>
  <Get the handle to incoming communication>
  <Pass the handle to handle_connection() in Web.pm>
</LOOP>

<Close socket>
```

Outline of handle_connection()

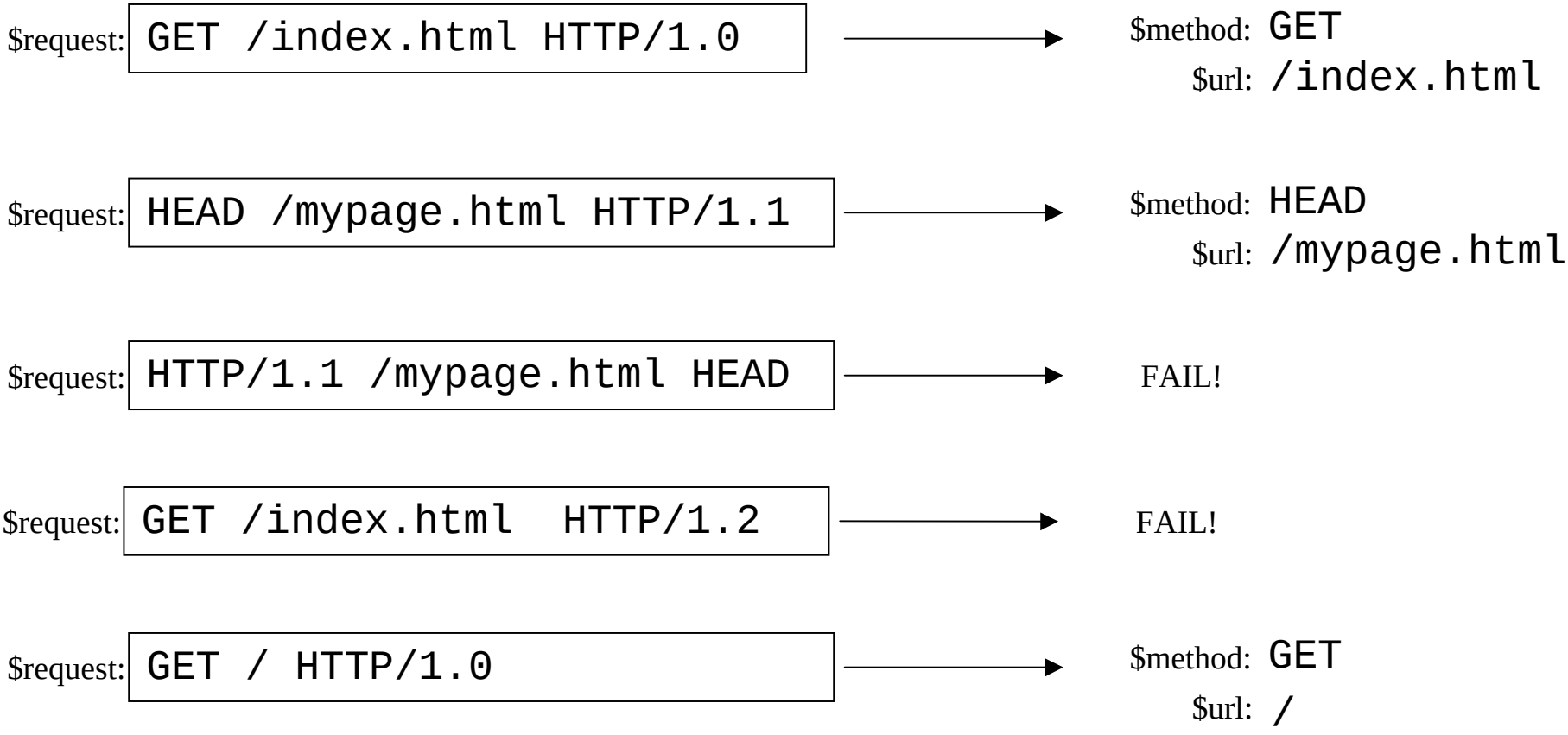
```
<Set the socket handle supplied as a parameter>
<Set the standard end-of-line character for HTTP messages>
<Read the contents from the socket handle into a request varible>

<CHECK FOR ERRORS SECTION>
  <Check to make sure the main request line has the right string
       format. Call invalid_request() otherwise. Set $method to
       GET or HEAD, and $url to the supplied URL>
  <Call lookup_file() to find the specified $url in the file system.
       Call not_found() if lookup_file() fails.>
  <If the type of 'file' return from lookup_file() is actually a
       directory, call redirect()>

<Print the status line and the headers for the response to the
socket handle (ie. to the client)>

<If the HTTP method is "GET", print the file requested in the URL to
the socket handle (ie. to the client)>
```

Checking the request line:



Checking the request line:

- Some examples:



Outline of lookup_file()

```
<Set the URL supplied as a parameter>

<Add the DocumentRoot of your web site to the front of the URL -
       converting from the URL to the file path>

<Get rid of spurious characters in the file path.>

<Fail if the the file path is really a path to a directory>

<Determine the file type and size>

<See if the file can be opened>

<return the a handle to the file, the file type and size>
```

Outline of invalid_request(), redirect() and not_found()

```
<Print to the socket handle (ie. to the client) the appropriate
headers for the HTTP response.>

<Print the appropriate message in HTML to show to the user at the
client side.>
```

- These subroutine sends the relevant responses to the web client when there are errors.

What's missing for Assignment 1?

- The only thing you need to implement in Assignment 1 pat II that is missing from the above code is the dealing with the special files with extension .336
    - It's that simple!
    - But remember you must show that you understand what you submit – that's not too much to ask.