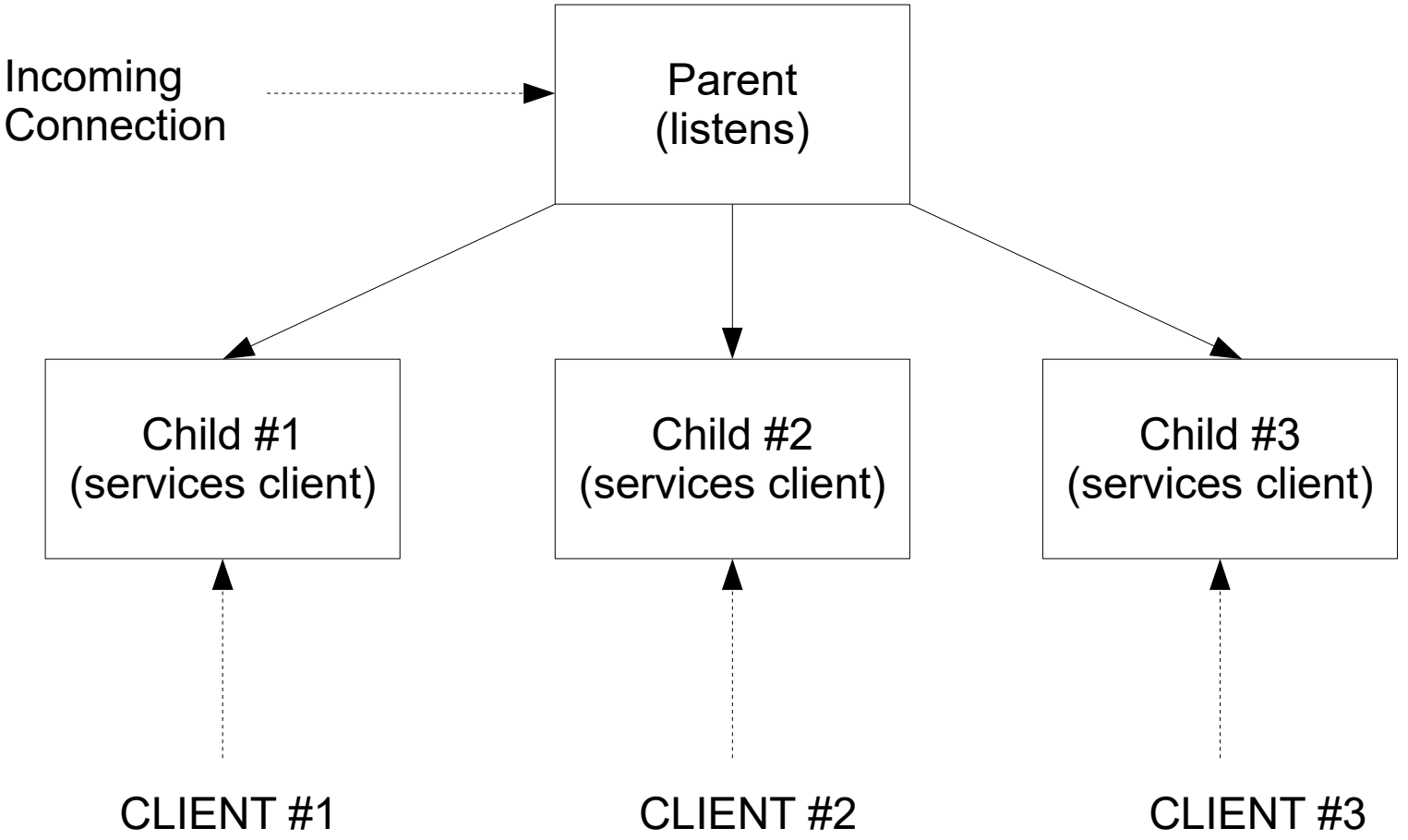
# Concurrent TCP Servers

CIS-3152, Network Programming
Vermont Technical College
Peter C. Chapin

# Concurrency Necessary

- Most services require concurrent servers.
  - Clients may require a “long time” to service.
    - Long downloads
    - Multiple commands
  - Clients might connect and do nothing.
    - Because they are broken
    - Because they are malicious
  - Network might be slow
- Can't afford to block other clients!

# Process Tree



# Multi-Thread Alternative

- Create a thread for each client instead.
  - This is good because...
    - Thread creation faster than process creation
    - Easy for threads to share resources
  - BUT...
    - Less isolation between threads than processes
    - Multi-threaded programming is tricky.
- We will focus on process level concurrency.

# Unix `fork` Function

- Once a connection has been accepted...
  - ```
    if ((child_ID = fork()) == -1) {
      perror("Unable to fork");
      return error_code;              // Is this right?
    }
    else if (child_ID == 0) {         // We are the child.
      close(listen_handle);           // Don't need this.
      // Service connection...
      close(connection_handle);       // Close connection.
      exit(0);                        // Child terminates!
    }
    ```
  - `fork` creates an identical copy of the parent.
    - Both parent and child run the same code!
    - Returns child process ID to parent.
    - Returns zero to child.
    ```

# Parent's Main Loop

- The parent accepts connections and forks a child for each...
  - ```
    while (1) {
      if ((connection_handle = accept(...)) == -1) {
        perror("Accept failed");
      }

      // Create child to service client (previous slide)

      // Parent doesn't need this handle.
      close(connection_handle);
    }
    ```
  - Parent calls accept again “as soon as possible”
    - Next client doesn't have to wait.
    - NOTE: *Child inherits parent's handles!*
    ```

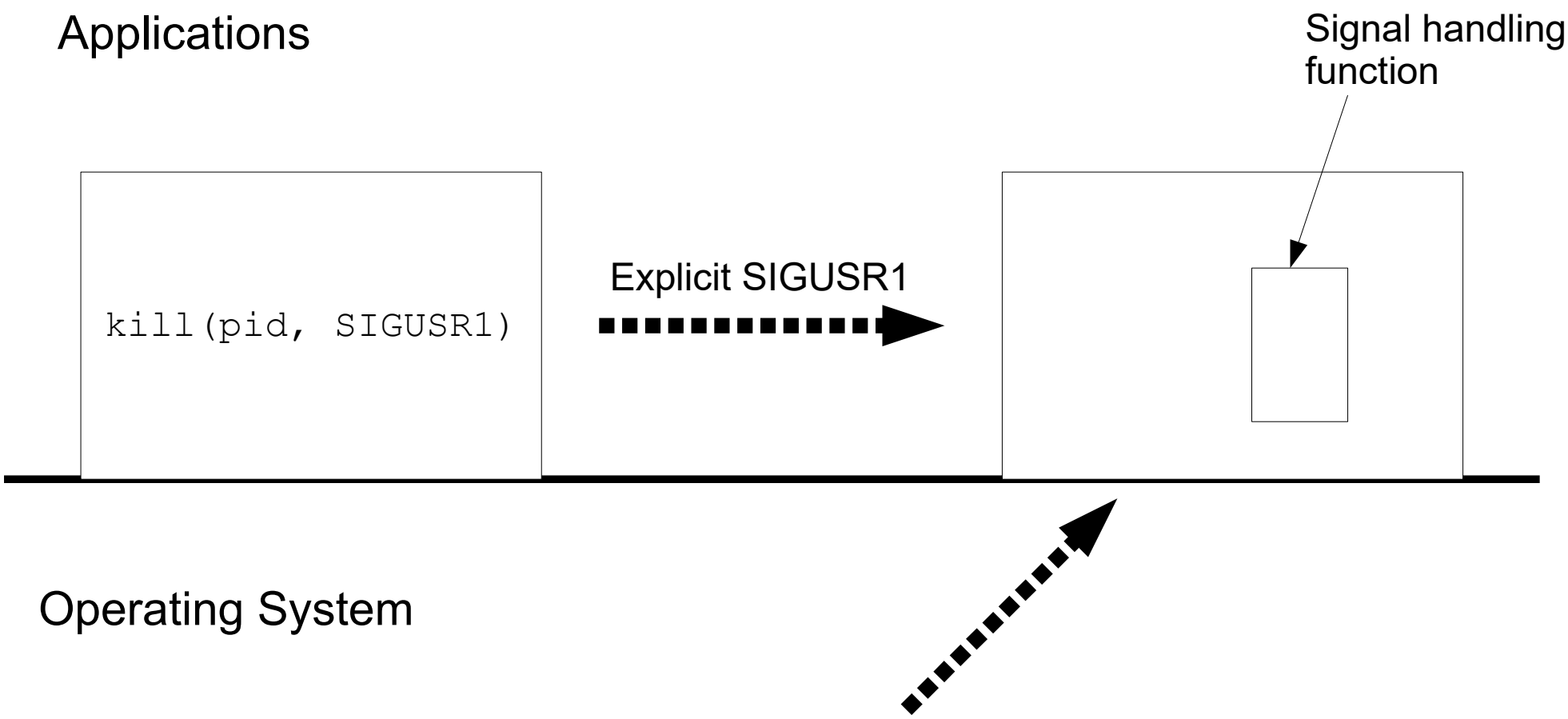
# Zombies

- Each process produces an “exit status” to return to its parent.
  - Can be used to signal success/failure.
  - When a process terminates it becomes a “zombie” until parent reads its exit status.
  - Unless the parent server handles this, zombies will accumulate.
  - Zombies are also called “defunct” processes.
- Previous code did not deal with zombies.

# Signals

- A “signal” is a software interrupt.
  - Unix-specific concept.
    - Windows does things differently.
  - Generated by the operating system.
  - Many different system signals defined.
- When a signal is received...
  - The process might be killed.
  - The signal might be ignored.
  - A special “signal handling function” might be called.
  - *Action depends on signal and on program.*

# General Structure



# SIGCHLD

- The `SIGCHLD` signal indicates child termination.
  - Unix sends the parent `SIGCHLD` when one of its children dies.
  - Normally `SIGCHLD` is ignored.
  - We must...
    - Install a signal handling function for `SIGCHLD` that:
    - Collects the exit status of the child
      - Eliminate the zombie!

# Set Up Signal Handling

- During the program's initialization...
  - ```
    struct sigaction action, old_action;

    action.sa_handler = SIGCHLD_handler;
    sigemptyset(&action.sa_mask);
    action.sa_flags = 0;
    sigaction(SIGCHLD, &action, &old_action);
    ```
  - `SIGCHLD_hander` is a pointer to the signal handling function (defined elsewhere in your program).
  - `sigaction` installs the new handler and returns the old handler information.
  - See the man page for more details.
    ```

# SIGCHLD Handler

- Also need an appropriate function for handling the `SIGCHLD` signal...
  - ```
    void SIGCHLD_handler(int signal_number)
    {
      int status;

      while (waitpid(-1, &status, WNOHANG) > 0) ;
    }
    ```
  - Called whenever `SIGCHLD` received.
    - Uses `waitpid` to retrieve the exit status of a child.
    - Loops to handle all dead children
      - Multiple children might have terminated “at the same time.”
    ```

# Slight Complication

- Blocking system calls (like `accept`) return “spuriously” after a signal has been handled.
  - This gives your application control again.
    - You might want to do something different.
  - In our case, we just want to call accept again.
    - When a child dies we just want to go back to what we were doing (waiting for a new connection).

# Call accept In a Loop

- Instead of a simple conditional statement...
  - ```
    while ((connection_handle = accept(...)) == -1) {
      if (errno != EINTR) {
        perror("Accept failed!");
        return error_code;
      }
    }
    ```
  - `accept` returns with `errno` set to `EINTR` if it is "interrupted" by a signal.
    - This is not really an error!
    - Code above just ignores that case and calls `accept` again.
    ```

# Other Possibilities

- Some Unixes allow you to...
  - Set a flag in the `sigaction` structure so that system calls are automatically “restarting”
    - No `EINTR` return.
  - Set a flag in the `sigaction` structure so that dead children don't create zombies in the first place.
  - Features are optional according to POSIX.
- Does Linux allow either of this options?