1 wait()

Wait is used to wait for a change of state in a child process, and to obtain information about the child process whose state has changed.

A state change is considered to be:

- The child terminated
  - In this case, performing a wait allows the system to release the resources associated with the child who terinated. **If a wait is not performed, the child is considered to be a "zombie"**
- The child was stopped by a signal
- The child was resumed by a signal

If the child has already changed states then a call to wait will return immediately. Else they will block until either a child changes state or a signal handler interrupts the call.

2 How2Shell

How a shell works (generally):

- A shell is just another program.
- What happens when a user runs a program:
  - Based on the `PATH` environment variable, the shell will search for binaries that correspond to the user's requrest.
  - If the user specified output redirection, the shell will redirect STDOUT / STDIN appropriately (see section on this)
  - Fork the current process, the child proccess will execute the specified program.
  - The child process sends a signal to the parent process (shell), shell continues execution.

There are exceptions for I/O redirection, piping, and running programs in the background, however the above workflow applies to all of those cases.

3 Working Dir

The working directory represents where your program, or shell, currently is.

- To change the current working directory in bash, use the `cd` command.
- To change the current working directory in C, use the `chdir()` function.

4 Globbing and Wildcards

Globbing and wildcard matching is something done by the shell, with logic implemented by the C library `glob.h`, or the logic can be defined manually. It dosen't really matter as long as it behaves like a normal glob

4.1 How To Glob

Consider the following files:

```
book1 book2 book3 potato1 potato2 potato3
```

To get all books, a few possible wildcard statements would be `b*`, `book*`, etc. If I wanted to get all things with a 3 I could do `*3`. If I wanted to get everything containing the letter o and the number 2, I could do `*o*2`
Globbing isn't understood by a program, so all of these glob statements **expand** out to be a list of files, i.e

```
b* = book1 book2 book3
```

5 I/O Redirection

The I/O redirection we implemented is *ms-dos* style. Because dos could only run one process at a time, the general workflow was something like this

1. Run process A and put the output to some TMP file `A > tmpfile`
2. Run process B with the input redirected from the tmpfile `B < tmpfile`

In a more modern multiprocess system this would not involve writing anything to the disk – however here is how we implemented this in 3000shell:

1. Run process A and put the output to some TMP file `A > tmpfile`
   (a) **Run process A:** fork the current process, wait for the child to finish.
   (b) **Put the output to some TMP file:** `open` the tmpfile, get the file descriptor. Create a copy of the old file descriptor, redirect stdout to the output file.
2. Run process B with the input redirected from the tmpfile `B`
   (a) **Run process B:** fork the current process, wait for the child to finish.
   (b) **Put the output to some TMP file:** `open` the tmpfile, get the file descriptor. Create a copy of the old file descriptor, redirect stdout to the output file.

6 Umask

Umasks, or file mode creation masks, are groupings of bits that define permissions for files.
First, consider the model that is used for file permissions:

- User – permissions for the immediate user
- Group – permissions for users that are in a files group
- Others – permissions for all other users

For each class there corresponds a set of bits that represent the actions that can be applied to the file:

- r – read access to the file.
- w – write access to the file.
- x – permission to execute the file.

These permissions values can be represented as three bits, in the order RWX. For instance to specify read, write, and not execute, the corresponding value will be 110.

Translation to octal: each set of these bits is effectively an octal value (base 7). So all of these binary values can be represented as octal, for example

- RWX-RWX-RWX = 111 111 111 = 777 permissions
- RW–RW–R– = 110 110 100 = 664 permissions

7 System and Library Calls

System calls are one layer above the kernel – all a system call does is asks the kernel to do something. They are very low level. Library calls consist of system calls, so library calls can be thought of as an *abstraction* of system calls

```
     Applications
     ------------
Shell   Library Routines
-------------------------
    System Calls
    ------------
       Kernel
```

8 TTY's And Proc

TTY's are text input / output environments. Typically these

TODO: Add more here

9 Signal Handlers

9.1 How To Define and Register

Typically a signal handler is defined in a method, and looks like this:

```c
void signal_handler(int the_signal)
{
  int pid, status;

  if (the_signal == SIGHUP) {
    fprintf(stderr, "Received SIGHUP.\n");
    return;
  }

  if (the_signal != SIGCHLD) {
    fprintf(stderr, "Child handler called for signal %d?!\n",
            the_signal);
    return;
  }

  pid = wait(&status);

  if (pid == -1) {
    /* nothing to wait for */
    return;
  }

  if (WIFEXITED(status)) {
    fprintf(stderr, "\nProcess %d exited with status %d.\n",
            pid, WEXITSTATUS(status));
  } else {
    fprintf(stderr, "\nProcess %d aborted.\n", pid);
  }
}
```

In the above example we have a generic method that takes an int (since that's all a signal is) and then we provide logic depending on the signals we want to handle.

Signal handlers are typically registered when a program starts up, in the main method.

```
int main(int argc, char *argv[], char *envp[])
{
  // A struct for signal handlers
  struct sigaction signal_handler_struct;

  // Fill the struct with 0s
  memset (&signal_handler_struct, 0, sizeof(signal_handler_struct));

  // Assuming we have a signal_handler method
  signal_handler_struct.sa_handler = signal_handler;

  // Provide  behavior  compatible with BSD signal semantics by making
     certain system calls restartable across signals.
  signal_handler_struct.sa_flags = SA_RESTART;

  // Now we can register our signals to our signal handler.
  if (sigaction(SIGCHLD, &signal_handler_struct, NULL)) {
    fprintf(stderr, "Couldn't register SIGCHLD handler.\n");
  }

  if (sigaction(SIGHUP, &signal_handler_struct, NULL)) {
    fprintf(stderr, "Couldn't register SIGHUP handler.\n");
  }

  // ... rest of your program
}
```

10 UNIX Pipes

The unix pipeline redirects the standard output of one program to the standard input of another program.

NOTE in assignment 1 we implemented a MS-DOS style pipe command which is different from what happens in UNIX. MS-DOS, because it could only run one program at a time, would do the following

- Run program A, redirecting stdout to a temporary file `A > tmp.txt`
- Run program B, redirecting stdin from a temporary file `B < tmp.txt`

- This would happen seamlessly when the user types `A | B`

11 Hard Links vs Symlinks

11.1 Hard Links

A hard link can be thought of as a second filename. For example, inode *x* can have filename `blah.txt` when it's created, but if I hardlink `blah.txt` to `notblah.txt` I'm just updating that inode's record with two names: `blah.txt` and `notblah.txt`

Hard linking is done by `ln source dest`. A hardlink appears as a copy of a file, but editing it edits both filenames (since they link to the same inode).

Hard links, because of how they're created, are hard to tell from regular files (since they are also regular files). In the below example I created file.txt, and ass.txt which hardlinks to file.txt

```
$ ls -al
total 8
drwxr-xr-x 1 conner users 30 Oct 12 16:08 .
drwxr-xr-x 1 conner users 52 Oct 12 16:07 ..
-rw-r--r-- 2 conner users 15 Oct 12 16:10 ass.txt
-rw-r--r-- 2 conner users 15 Oct 12 16:10 file.txt
```

Hmm, these appear to be the same. A simple way to tell if a hardlink exists is to look at the **inode** that it corresponds to. This can be done with `ls -i`

```
$ ls -li
total 8
1471517 -rw-r--r-- 2 conner users 15 Oct 12 16:10 ass.txt
1471517 -rw-r--r-- 2 conner users 15 Oct 12 16:10 file.txt
```

Because the inodes for those two files are the same, we know that they are hardlinked together.

11.2 Soft Links

A soft link, or symlink, can be thought of as a pointer to an inode. The difference between a symlink and a hard link is that this **not** another name added to an existing file, rather, it's a new file that points to the inode of another file. This means that the source file has to exist in order for the symlink to be valid.

Symlinks are created by `ln -s source dest`, and dest will have a new inode that points to the source file. See below for an example

Here, I created `link` which points to `file.txt`

```
$ ls -al
drwxr-xr-x 1 conner users 24 Oct 12 16:16 .
drwxr-xr-x 1 conner users 52 Oct 12 16:07 ..
-rw-r--r-- 1 conner users 15 Oct 12 16:10 file.txt
lrwxrwxrwx 1 conner users  8 Oct 12 16:16 link -> file.txt
```

We know right away that it's an symlink because of the arrow in ls, and we can verify the symlink resides on another inode from `ls -i`

```
$ ls -li
total 8
1471517 -rw-r--r-- 1 conner users 15 Oct 12 16:10 file.txt
1471741 lrwxrwxrwx 1 conner users  8 Oct 12 16:16 link -> file.txt
```

12 Object Files vs Executable Files

When a source file gets compiled, an object file gets created. This object file is mostly ready to be ran, however, there are unresolved references to external libraries that need to be included. After these references have been resolved, we have a binary thats not useless!

13 3000Copy Sequence of Operations

In all cases, we perform the following initalization steps:

1. Parse the source and destination args
2. Open the source and dest files, get the file descriptors

13.1 RW Implementation

```
done = 0;
while (!done) {
        len = read(source_fd, buf, BUFSIZE);
        if (len == -1) {
                report_error("reading source", strerror(errno));
        }

        if (len > 0) {
                if (write(dest_fd, buf, len) == -1) {
                        report_error("writing dest", strerror(errno));
                }
        } else {
                done = 1;
        }
}
```

1. Read in a maximum of BUFSIZE bytes into our buffer, take note of how many bytes we read
2. If we have bytes, write them to the destination file
3. Else we are done! Close file descriptors.

13.2 MMAP Implementation

```
        len = statbuf.st_size;
        source_map = (char *) mmap(NULL, len,
                                   PROT_READ, MAP_SHARED, source_fd, 0);

        if (source_map == MAP_FAILED) {
                report_error("source map", strerror(errno));
        }

        if (stat(dest_fn, &statbuf) == -1) {
                printf("%s does not exist, creating\n", dest_fn);
        } else {
                printf("%s exists, overwriting\n", dest_fn);
        }

        //... repeat the above for the dest FD

        printf("Copying bytes...\n");

        for (i = 0; i < len; i++) {
                dest_map[i] = source_map[i];
        }
```

1. Open the source and dest file, get the file descriptors.
2. Get the length of the source file. Create a mapping the size of the source file into virtual memory using the FD we got from opening the source file.
3. Get the length of the dest file. Create a mapping of the size of the dest file into virtual memory using the FD we got from opening the dest file.
4. Iterate over the source map, transferring its values to the destination map.
5. Close the file descriptors

14 Assembly RBP, RSP

`RBP` is the base pointer, and `RSP` is the stack pointer.

- `RSP` memory address at the top of the stack
- `RBP` memory address of any stack frame, moves around the stack.

15 I/O

15.1 File I/O

- `open / openat`, the `open` system call opens a file, if the file does not exist it will be created if the `O_CREAT` flag is set.
- `read`, system call to read bits from a file descriptor
- `write`, system call to write bits to a file descriptor
- `lseek`, system call to reposition the file offset of an open file descriptor
  - lseek past the end of the file and write for a good time
- `mmap`, map or unmap files into memory, mmap creates a mapping into the virtual address space of the calling process.
- `close`, ass.

15.2 Directory I/O

- `opendir/open`, opens a directory, returns a `DIR` stream / struct.
- `readdir/getdents64`, readdir is older and is superseded by getdents64, used for getting a directories entries.
- `closedir/close` self explanitory.

16 `mmap`

What is this for??

From what we've used it for, file-backed mapping. If I had a file of size *n* and I wanted to read it, my general algorithm would be allocate a buffer to read the file, read file, play with file, life is great. However, if my file is size *n* and my memory is size *k* where $n > k$, we have a problem.

mmap is handy because we never have to malloc anything - mmap will "map" the file into memory without explicily reading it. Now, when you play with that memory area `mmaped_file[i]`, those memory operations will **transparently** act on the file. It lets you map files into memory that are larger than the available memory on the system using the OS's available paging mechanism.

There are a few ways of using MMAP:

- `PROT_EXEC` – used for loading executable code
- R/W mmap, shared – used for writing to and modifying files on disk
- R/W mmap, private – used for copying the file into memory and modifying the memory
- R/W mmap, anonymous + shared – used for allocating memory that's shared between processes
- R/W mmap, anonymous + privte – allocate memory to this process.

17 Process Memory Map

From the lowest to the highest address:

- **Text segment**: contains executable instructions, this is placed below the heap or stack in order to prevent heaps and stack overflows from overwriting it.
- **Initialized data**: above the text segment, contains global variables and static
- **Unitialized data / BSS**: BSS = block started by symbol. Data in this segment is initialized by the kernel before the program starts executing.
- **Heap**: begins after the BSS segment, grows upwards towards the stack. Data in the heap is managed by malloc et al.
- **Stack**: Sits at a high address, grows down towards the heap. the stack holds automatic variables, and *stack frames* – data about function calls and contexts. In general, we want to use the heap more than the stack because the stack ideally is used for execution-level stuff (frames)

Sample Questions

1. What system call is used to load a progam binary in Linux? Does this system call create a new process?
   `execve`, and no – `fork` is used to create a process.

2. Dynamially linked programes generally make many mmap calls at the start of program execution that a statically linked version of the program dosen't do. What are those mmap system calls for?
   Because the programm is *dynamically linked* the mmap calls are to map libraries into the process's address space

3. Symbolic links map filenames to filenames. What do hard links map filenames to?
   inodes

4. What are two standard uses of signals in Linux? Do not list application specific signals like `SIGUSR1`
   Some examples that can be used are `SIGTERM, SIGKILL, SIGINT, SIGCHILD`.

   `SIGINT` – keyboard interrupt, `SIGTERM` – termination signal, `SIGKILL` – kill signal (most deadly), `SIGCHILD` – child stopped or terminated.

5. Do pointers in C contain virtual or physical addresses? Why?
   Pointers contain virtual addresses, because the kernel abstracts the physical memory and users only have access to virtual memory

6. Does a process make a system call to allocate memory? Why?
   A process must make a system call like `mmap()` in order to allocate memory. Memory is a physical resource, and the kernel controls all system resources, including the virtual-to-physical mapping of memory address spaces. Because of all of this a system call has to be made, since system calls are how we talk to the kernel.

7. At the prompt of a shell, a user types `something > output.txt`. What programs the file `output.txt` – the `shell`, or `something`? Why?
   `shell` opens `output.txt`. The shell will parse the prompt string, get the filename, open the file and gets a file descriptor for it.

   After forking, the child process that shell forked will replace it's own file discriptor for `STDOUT` with the output file's FD with a function like `dup2()`. Finally, the child process will execve `something` and then the modified FD for `STDOUT` will persist through the call to `execve()`

   `execve()` preserves file descriptors so a running program can do I/O in a generic way that allows programs to be combind together into larger solutions.

8. Is it possible for two or more filenames to refer to the same inode? Explain.
   HARD. LINK.

9. How can a process send a signal?
   By using the kill system call.

10. How can a process change what happens when a signal is received?
    By registering a signal handler for a particular signal, this is done with a sigaction system call.

11. Can a process change what happens when any signal is received?
    No, some signals such as KILL and STOP have predefined signal handlers that can't be overwritten.

12. When a process receives a signal, what happens to the function that the process was already running?
    The function is paused and its execution context is pushed onto the function call stack, the signal handler fucks shit up, and it resumes from where it left off once the signal handler finishes.

13. Consider the following code from 3000shell:

```c
if (background) {
  fprintf(stderr, ‘‘running in background’’);
} else {
  pid = wait(ret_status);
}
```

(a) What does 3000shell.c call before this in order to create a child process?
`fork()`

(b) What does the call to wait() above do?
It waits for the child process to exit, it returns the PID that exited, and ret-status is a variable being passed-by-reference, so if the return status is non-null wait will put it into the ret_status variable

(c) When does 3000shell also call wait()?
It also calls wait in the SIGCHLD signal handler. The kernel sends 3000shell a SIGCHLD signal when a child process terminates. The signal handler code is necessary to take care of background processes (i.e., prevent them from becoming zombies).

(d) Does the process that executes the above code also run the program the user has typed in at the command prompt? Explain briefly.
It does not. The above code runs in the main 3000shell process. Commands are execve'd in child processes.Only the child processes can call execve because otherwise 3000shell would effectively terminate by running execve.

14. MMAP is declared as follows:

```c
void *mmap(void *addr, size_t length, int prot, int flags,
                     int fd, off_t offset);
```

(a) At a high level, what does mmap do?
mmap associates (or maps) the contents of a file with a range of memory addresses without reading the entire file into memory – very cool stuff! If we don't specify a file it will just allocate us memory to fap around with.

(b) What's the difference between setting flags to be MAP_SHARED versus MAP_PRIVATE in the context of child processes?
With MAP_SHARED, all child processes will share the same memory, changes made by one will be visible to all. Also, if a file descriptor was specified, its contents will be changed when the corresponding memory is changed. With MAP_PRIVATE, all changes to memory are private. Every process has its own unique copy (with its contents being copied on fork like all other regular memory). Further, the opened file is not changed to reflect the changes to the mapped copy.

(c) If we open a file and mmap it, setting prot=PROT_READ—PROT WRITE and flags=MAP_PRIVATE, can we modify the memory that is returned by mmap? If we modify this memory, will it modify the file as well? Explain briefly.
If we set PROT_READ and PROT_WRITE, we can indeed modify the returned memory. Any changes, however, will be private if MAP_PRIVATE is specified, so the opened file will not be changed; the initial contents of memory will be the same as the file, but subsequent changes will be private to the process.