1) Two basic principles/metaphors of Unix are File and Process.

2.1) An i-node is something used to represent a file; an i-node stores information about the file on the disk.

2.2) 12 direct blocks which contain blocks of data

3 indirect blocks

Single indirect blocks point to addresses that contain data

Double indirect blocks point to addresses that contain addresses that contain blocks of data

4GB will only use Double indirect blocks with how tree spans

3.1) Relative path specifies how to get to a file from the current directory while absolute path specifies how to get to a file from the root directory.

3.2) The absolute path of the $APUE_HOME directory is /export/homes/eberta/csci3751/apue.3e

3.3) The relative path from ~/tmp is ../csci3751/apue.3e/

4.1) Zombie processes can be created when a parent process doesn't wait on the child to terminate, and thus doesn't receive an exit status, which means the terminated process still exists in the process table. This can be serious because too many zombie processes can fill the process table as it is only finite.

4.2)

```
[eberta@csci-gnode-02 ~]$ ps -eaf
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0  2023 ?        00:48:00 /usr/lib/systemd/systemd --switched-root --system --deserialize 22
```

5.1) PS1

5.2)
```
[eberta@csci-gnode-02 ~]$ echo $PS1
[\u@\h \W]\$
```

5.3) /etc/bashrc

6.1) Umask is a way of working backwards when setting file permissions. For example, the default permissions is 755 which would be umask 022 because 777 – 755 = 022

6.2) umask 016

7)
```
[eberta@csci-gnode-02 ~]$ find $APUE_HOME -name "*.c" -exec grep -H "fork(" {} \;
```

8)
```
:%s/apple/orange/g
```

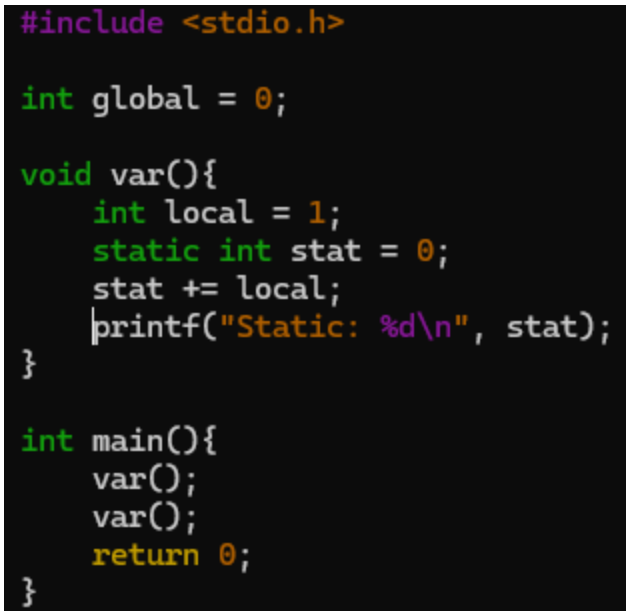
9) -g: debug mode: run/r creates a process that runs the program and will finish if no errors, otherwise it will display the line number the program stopped at with an explanation (best used when debugging errors within the program)

-o: optimization: you might want to use this when the final version is ready as this makes file executable where gcc -o exec_name file.c

-c : object code only : this might want to be used when manual linking occurs rather than an automatic linking

10.1) Global variables are located outside of a programs functions and thus accessible to all functions. Local variables are declared within a function and are thus accessible only within that function. Static variables are declared within a function but aren't destroyed on exiting a function and thus last the lifetime of the program.

10.2)
```
[eberta@csci-gnode-02 ~]$ ./test2
Static: 1
Static: 2
```



```c
#include <stdio.h>

int global = 0;

void var(){
    int local = 1;
    static int stat = 0;
    stat += local;
    printf("Static: %d\n", stat);
}

int main(){
    var();
    var();
    return 0;
}
```

11.1) if (val & O_APPEND)

11.2) val |= newVal

11.3) val &= ~newVal

12.1) Atomic operations involve all steps of a task being performed or none at all, this can create a problem, if different processes are writing to the same file at the same time, which can lead to undefined behavior. So, an all or nothing mentality works around such issues.

12.2) A possible solution to the example given above would be a wait or pause to make sure different processes aren't creating a race condition.

13) The first command will redirect stdout to outfile and then will redirect stderr to stdout which is directed to outfile as well. The second line will redirect stderr to stdout which is the terminal, and then will redirect output to outfile. So, the first one will have stdout and stderr directed to outfile while the second one only has stdout directed to outfile.

14.1) set-user-id is saved by exec function so that the effective user ID of process is set upon file execution.

14.2) Process determines file permission unless set-user-id is set then UID is temporarily set to st_uid, saved-set-user-id is saved and hold's process's original UID which is later set back to process's UID.

15.1) Some differences are concept(same inode as file vs shortcut file), command(ln vs ln -s), if original file is deleted(still valid vs invalid), and inode number(same vs different).

15.2) Soft links can be convenient if you are creating shortcuts to a lot of files, which contain an address vs an actual copy.

15.3) 180 soft links

16) Any buffered output data is flushed before the file is closed and any input data buffered is discarded.

17.1) You can tell by the return values, child pid for child or 0 for parent. Likewise, the PID and PPID can indicate parent vs child.

17.2) Inherited:

Real user ID, real group ID, effective user ID, effective group ID
Supplementary group IDs
Process group ID
Session ID
Controlling terminal
The set-user-ID and set-group-ID flags
Current working directory
Root directory
File mode creation mask (umask)
Signal mask and dispositions
The close-on-exec flag for any open file descriptors
Environment
Attached shared memory segments
Memory mappings
Resource limits

Not inherited:

The return value from fork (child pid vs 0)

The process IDs are different

The two processes have different parent process IDs:

- The parent process ID of the child is the parent; the parent process ID of the parent doesn't change

The child's tms_utime, tms_stime, tms_cutime, and tms_cstime values are set to 0 (process times)

File locks set by the parent are not inherited by the child

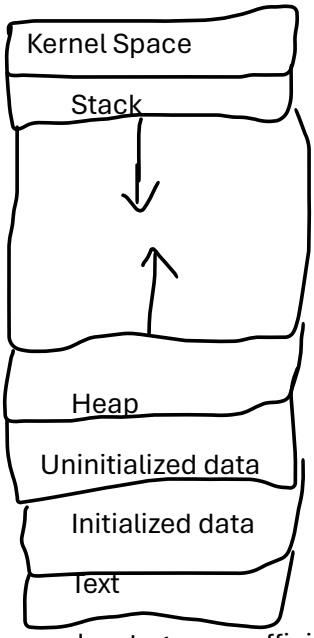
Pending alarms are cleared for the child

The set of pending signals for the child is set to the empty set

18.1) Seconds since the UNIX Epoch

18.2) On 32 bit systems the largest value that can be recorded is in 2038, however many systems work on 64 bit now which extends this end time

19)



high add

low add

20) Some advantages are efficiency over copy-on-write and it guarantees that the child will run first, some disadvantages are deadlock possibilities which could lead to program bugs

21.1) The child is attempting to wait for the parent process to terminate to avoid a race condition

21.2) It's bad practice because of polling(wasting CPU time)

22.1) Race conditions are when multiple processes are sharing data, so we can't know the result since order of execution is unknown.

22.2) Some ways around this is using Signal and IPC

23) SIGKILL and SIGSTOP can't be ignored because they are handled by the kernel in case of needing to exit; if there was no way to exit then it would continue indefinitely.

24.1) Slow system calls

24.2) System V and BSD

25.1) One signal is handled and the others are discarded

25.2) They are all handled, but not delivered in any fixed order

26.1) The first argument passed is the descriptors we're interested in, the next three are the conditions we're interested in, and the final is how long we will wait. The return tells us how many descriptors are ready, as well as which ones specifically

26.2) EOF is generated

26.3) SIGPIPE is generated

27) It has a name and can be used for communication between unrelated processes

28) close(fd[0])

write(fd[1]...)

close(fd[1])

read(fd[0])

30) mkfifo fifo1

wc -l fifo1

ps -le | tee fifo1 | sort -k13

1) c
2) a
3) d
4) c
5) d

6) b
7) b
8) d
9) d
10) e