CSEN177, Winter 2025 Homework 2

1. The transitions missing are reading to waiting, and waiting to running. Ready to waiting can happen if the OS misplaces a process and it has to wait to get processed by the CPU. Waiting to running happens if the task at hand is a high priority item so it skips the ready phase to run right away
2. Web browsers could od multiple tasks at once which is the main purpose of threads to share resources to allow for better responsiveness. In the case of a web browsers, using threads can allow for the site to render, fetch data, and handle user inputs at the same time with them blocking each other. In order to run multiple tasks at once the browser must be multithreaded so that it can run these simultaneously otherwise a single threaded browser won't have any benefits.
3. There are a few cases when a single threaded browser would be more beneficial. One would be if there was a lack of resources, thus using a single thread will have a lower cost to run. Another reason could be if there is a low workload/no reason for multiple processes to run at once. Thus, users cannot scroll as their search loads, or there is only one active user at a time with basic features, these would not necessarily need multiple threads as their costs and computational time will be low enough that there will be a minimal gain to multiple threads.
4. Benefits of thread yielding would be allowing for improved responsiveness. For example if the user is loading one frame they can then be yielded the CPU so that they can load in user input before moving on, providing a better and smoother experience for the user. One very important reason that could also be important is avoiding busy waiting, or a situation in which the CPU cycles in a loop, this process of thread yield will allow for that process to stop allowing for the CPU to work on more important tasks or even stop an endless loading cycle.
5. The biggest benefit to implementing threads in a user space is additional performance. For example, this happens as thread operations will be faster as they don't need system calls or Kernal intervention, and can run everything in user mode. This also allows for user libraries to implement their own scheduling algorithms, which can help optimize for specific use cases. The biggest disadvantage is that you can encounter blocking issues. For example is a user space thread makes system calls it can get blocked as the kernel won't know about the multiple threads and treat it as one.
6. There is a way to make the order to be strictly thread 1 created thread 1 prints message, thread 1 exits, thread 2 created, thread 2 prints message, thread 2 exists, and so on. This is by doing two things the first is using a shared variable to track when a thread needs to be ran, and then using condition variables to allow threads

to run when the expected behavior has been met, for example only run thread 2's sequence once thread 1 is created, prints messages, and exits.

7. Given the initial value was 5, by the time the code reaches Line A it has updated the child process to 20 due to the code, however the parent process will remain as 5 as the child does not modify the parent. Thus the output will be "PARENT: value = 5"
8. 2^n processes are made for n fork calls since there is 3 fork calls in the code there will be 8 total processes.
9. Init is the first process of booting in UNIX or Linux Systems and is responsive for initializing the system and starting other processes such as services or daemons. The more modern Linus distributions use system which is a more feature rich and modular system for this. These processes adopt and clean orphaned processes (processes in which the parent terminated but not eh child) and also do the same for zombie processes. Essentially, they clean up processes that would accumulate and waste memory to avoid unnecessary consumption.
10. At line A the output would 0, at Line B it would be 2603, at line C it would be 2603, and at line D it would be 2600.

11. Code files attached to submission

```c
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{

    int pipefd[2];

    struct timeval start, end;

    pipe(pipefd);

    if (fork() == 0)
    {
        close(pipefd[0]);

        gettimeofday(&start, NULL);

        write(pipefd[1], &start, sizeof(start));

        close(pipefd[1]);

        execvp(argv[1], &argv[1]);
    }
    else
    {
        close(pipefd[1]);

        read(pipefd[0], &start, sizeof(start));

        close(pipefd[0]);

        wait(NULL);

        gettimeofday(&end, NULL);

        printf("Elapsed time: %.6f seconds\n", (end.tv_sec - start.tv_sec) + (end.tv_usec - start.tv_usec) / 1e6);
    }

    return 0;
}
```

```c
#include <stdio.h>
#include <sys/time.h>
#include <unistd.h>
#include <sys/wait.h>
#include <sys/shm.h>

int main(int argc, char *argv[])
{
    int shmid = shmget(IPC_PRIVATE, sizeof(struct timeval), IPC_CREAT | 0666);

    struct timeval *start = (struct timeval *)shmat(shmid, NULL, 0);

    if (fork() == 0)
    {
        gettimeofday(start, NULL);

        execvp(argv[1], &argv[1]);
    }
    else
    {
        wait(NULL);

        struct timeval end;

        gettimeofday(&end, NULL);

        printf("Elapsed time: %.6f seconds\n", (double)(end.tv_sec - start->tv_sec) + (double)(end.tv_usec - start->tv_usec) / 1000000);

        shmdt(start);

        shmctl(shmid, IPC_RMID, NULL);
    }

    return 0;
}
```

```
[➜  COEN 177 ./hw2.11.1
Elapsed time: 0.000397 seconds
[➜  COEN 177 ./hw2.11.2
Elapsed time: 0.000223 seconds
➜  COEN 177
```