

CSC 4420 Assignment 1

Question 1 (3 points): What is the difference between timesharing and multiprogramming systems? Use your own words (do not copy and paste from slides!).

Time sharing systems involve users being allocated a portion of CPU time for user tasks. The CPU switches between these time intervals so quickly that it looks like the CPU is running each task at the same time. These frequent switches also allows the user to interact with a job while it is running. A time sharing system's focus is on allowing multiple users to interact with a CPU at the same time by focusing on faster response time.

Multiprogramming systems by comparison involves loading multiple programs into memory at the same time so that the CPU is always executing a job – there is not division of time, there are only jobs to be done. Multiprogramming aims to maximize the usage of the CPU.

Question 2 (3 points): Instructions related to accessing I/O devices are typically privileged instructions, that is, they can be executed in kernel mode but not in user mode. Give a reason why these instructions are privileged.

The privileges exist for accessing I/O devices so that users do not interfere with system critical hardware – this has the benefit of protecting the system from accidental and malicious actions such as interfering with system control registers or directly accessing memory that would otherwise compromise the stability of the operating system. I/O devices also require coordination and prioritization– something users cannot do as well as an operating system.

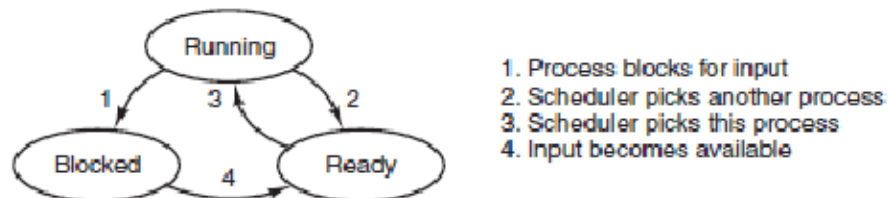
Question 3 (3 points): Cache coherency: Explain what it is and why the OS needs to ensure it.

Cache coherency is the consistency of data across different caches within an operating system. This is particularly important in systems with multiple processors where each processor has their own cache – if the shared data within those caches is not consistent then data corruption can occur when the processor attempts an operation relying on the cache and passes on the results when the cache is out of sync with the rest of the system.

Question 4 (3 points): On early computers, every byte of data read or written was handled by the CPU (i.e., there was no DMA). What implications does this have for multiprogramming?

It is significantly harder to build an efficient multiprogramming system when the CPU is handling every byte of data for operations such as I/O and memory management. This would mean that multiprogramming was extremely unresponsive and slow as the CPU was always busy.

Question 5 (3 points): In the following figure, three process states are shown. In theory, with three states, there could be six transitions, two out of each state. However, only four transitions are shown. Are there any circumstances in which either or both of the missing transitions might occur?



The conditions that are missing are from the ready state to blocked, and from the blocked state to running.

There are cases when a process is awaiting CPU time to be executed only for external circumstances to change meaning that it should be blocked from running. This can occur due to desynchronization, unavailability of resources, or a hardware interrupt.

There are cases where a process might transition from blocked to running, such as when an

interrupt occurs, or if the process is otherwise marked for the highest priority.

Question 6 (4 points): Using the following program, explain what the output will be at LINE A. Motivate your answer.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
    pid_t pid;
    pid = fork();
    if (pid == 0) { /* child process */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* parent process */
        wait(NULL);
        printf("PARENT: value = %d",value); /* LINE A */
        return 0;
    }
}
```

If we read over the code, this code creates a child process, it then checks the pid of itself, if it is the parent process, it waits for the child process to finish (see wait(NULL)), if it is the child process it increments the value by 15. FORKS CREATE COPIES OF THE PARENTS MEMORY SPACE MEANING THE 'VALUE' IN THE MEMORY SPACE OF THE PARENT IS UNAFFECTED AND THE RESULTING VALUE OF THE PARENT IS ONLY 5.

I typed out the code to test it because I don't trust myself.

```
#include <stdio.h>

#include <sys/types.h>

#include <unistd.h>
```

```
int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /*child process*/
        value += 15;
        return 0;
    }
    else if (pid > 0) { /*parent process*/
        wait(NULL);
        printf("PARENT: value = %d", value); /*LINE A*/
        return 0;
    }
}
```

Plugging this code into onlineGDB gave me a value of 5.

Question 7 (4 points): Given the code below, what are the values of pid/pid1 at lines A, B, C, and D? Assume that the parent's pid is 10 and the child's pid is 2.

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
int main(){
    pid_t pid, pid1;
    /* fork a child process */
    pid = fork();
    if (pid < 0) { /* error occurred */
        fprintf(stderr, "Fork Failed");
        return 1;
    }
    else if (pid == 0) { /* child process */
        pid1 = getpid();
        printf("child: pid = %d",pid); /* A */
        printf("child: pid1 = %d",pid1); /* B */
    }
    else { /* parent process */
        pid1 = getpid();
        printf("parent: pid = %d",pid); /* C */
        printf("parent: pid1 = %d",pid1); /* D */
        wait(NULL);
    }
    return 0;
}
```

This question is asking what the PID is of a child process and of the parent process after a fork occurs. Line A prints the child's initial id, which is 0 by default, line B prints the child's id which is 2, C is also 2 as it's the id of the child created, line D will give the parent PID of 10

Here is the code I slapped in so I could test it:

```
#include <sys/types.h>
#include <stdio.h>
```

```
#include <unistd.h>

int main(){

    pid_t pid, pid1;

    /* fork go brrrr*/

    pid = fork();

    if (pid < 0 ){ /*ERROR AHHHHH*/

        fprintf(stderr,"Fork Failed, We'll get em next time");

        return 1;

    }else if (pid == 0 ){ /* child process*/

        pid1 = getpid();

        printf("LINE A Child: pid = %d", pid); //LINE A

        printf("\n");

        printf("LINE B Child: pid1 = %d", pid1); //LINE B

        printf("\n");

    }else{

        //PARENT WORLD WOW

        pid1 = getpid();

        printf("LINE C Parent: pid = %d", pid); //LINE C

        printf("\n");

        printf("LINE DParent: pid1 = %d", pid1); //LINE D

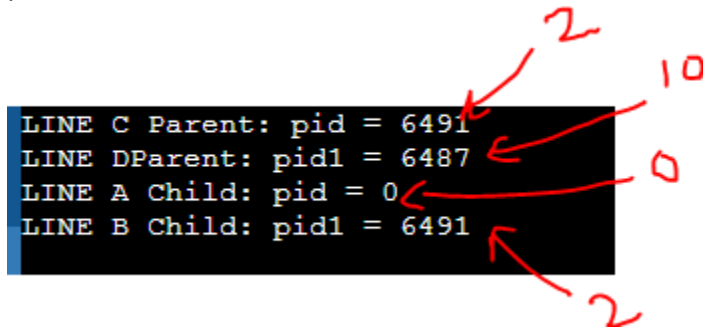
        printf("\n");

        wait(NULL);

    }

    return 0;
```

}



A screenshot of a terminal window showing four lines of program output. Red handwritten arrows and numbers are used to trace the execution flow. The first arrow points from the number '2' to 'LINE C Parent: pid = 6491'. The second arrow points from the number '10' to 'LINE DPARENT: pid1 = 6487'. The third arrow points from the number '0' to 'LINE A Child: pid = 0'. The fourth arrow points from the number '2' to 'LINE B Child: pid1 = 6491'.

```
LINE C Parent: pid = 6491
LINE DPARENT: pid1 = 6487
LINE A Child: pid = 0
LINE B Child: pid1 = 6491
```

Question 8 (2 points): Including the original process, how many processes are created by the program shown below?

```
#include <stdio.h>
#include <unistd.h>
int main(){
    int i;
    for (i = 0; i < 5; i++)
        fork();
    return 0;
}
```

Fork() creates a direct copy of the parent process. This results in an exponential growth scenario where there are 2^n processes for every iteration of the loop. Considering there are 5 iterations of the loop, this means there are $2^5 = 32$ processes created.

Here is a terrible sketch proving it because I don't trust my own math:

