# CS 159: Parallel Processing
# Project 1

Lab Background:

The primary purpose of this first programming lab for CS159 is to demonstrate a basic system call utility, *fork*, to create multiple processes from a single source code program. This will allow the student to experiment with the basic concepts of parallelism using a (hopefully) familiar and simple "C" programming environment. However, because of output buffering and scheduling optimization done by the operating system, some preliminary information is needed before the gist of the *fork* lab is discussed.

This preliminary section explains and demonstrates some of the printing anomalies that may occur during project 1 due to process race conditions and buffered I/O. The goal of the buffering provided by the standard 'C' I/O library routines is to use the minimum number of actual device-level read and write calls. Also, the OS tries to do its buffering automatically for each I/O stream, obviating the need for the application to worry about it. Unfortunately, buffering is the single most confusing aspect of the standard I/O library, especially when its actions are combined with a scenario where processes are being forked (as is done in this homework assignment).

'C' processes execute on top of the kernel (which is itself, a process). Output generated by a 'C' process with a *printf* statement is buffered by the kernel before being routed to the hardware; that is, output generated by the 'C' process is stored by the kernel to a temporary buffer until a sufficient amount of it has been accumulated to warrant disrupting process execution with an I/O fault. The system does this because it is more efficient for overall system performance to buffer I/O to minimize the number of I/O operations actually performed by the relatively slow physical devices (usually the disk). Recall that every time a running process does an I/O, it voluntarily gives up the RUN state and is moved to the WAIT state while waiting for the relatively slow I/O to be performed. This automatic, and seemingly ad-hoc and opportunistic optimization means that even though a process may have been coded by the programmer to print 5 bytes each time it goes through a 10-iteration loop, the system may in actuality perform just one write of 50 bytes after all 10 iterations through the loop have been completed. Typically, several *printf* statements can be executed by the process (and buffered by the kernel) before an actual write operation needs to be performed to a physical device or resource. This minimizes the number of transitions a process will need to make to the WAIT state.

The system generally performs buffering when it "knows" that the output is a disk (a file) rather than the terminal, since a person never really has the opportunity to see the output of the disk (file) until the process generating the output is complete and the file closed. Then, and only then, will the user typically open the file again for reading. It doesn't really matter if the file was written character-by-character, or all at once in a single burst upon process completion. Thus, the buffering is, in practicality, transparent to the user. For example, such a scenario occurs whenever the output of a process is piped to a file via the UNIX shell command " > ". Only after the program is complete will the user examine the output target file of the ">" shell command. The file appears the same to the user independent of whether it is written in real-time or buffered by the system.

When the output is directed to a terminal however, the system "knows" that there is a person sitting in front of the device expecting output to occur in real time. It then must actually perform the I/O operations as they occur. This way, if a user prints 5 bytes each time a program goes through a loop (a common debugging technique), (s)he actually sees 5 bytes print out on the screen each time the program executes through the loop, rather than seeing 50 bytes print out all at once after all 10 iterations through the loop have been completed. This may be important to the user because (s)he may be doing the incremental printing as a way of tracing the execution of the program. In fact, we will be doing exactly this as part of the lab assignment.

The differing requirements of output directed to a terminal vs. that directed to a file lead to the following buffering rules: A new line feed will generally force the system to actually perform the output if the output device is the terminal; however, it typically will not force an I/O to be performed if the output device is a disk (file). Error messages generated by the system need to be reported to the user as soon as possible and as close to the offending executable statement as possible. Therefore, error messages cannot be buffered at all.

Thus, there are three types of buffering provided by the system:

1) Fully buffered. In this case, actual I/O only takes place when the I/O buffer is filled. Files that reside on disk are normally fully buffered by the standard I/O library. The term "flush" describes the writing of a standard I/O buffer to a device. The buffer can be flushed automatically by the kernel, or manually by the programmer via a call to the function *fflush*. *Printf* behaves this way when directed to a disk.

2) Line buffered. In this case, the standard I/O library performs I/O when a newline character is encountered on input or output. This allows a program to output a single character at a time, but the actual I/O will take place only when the program finishes writing each line. Line buffering is typically used on a stream when it is directed to a terminal. *Printf* behaves this way when directed to a terminal.

3) Unbuffered. Here, output generated by a program is written to the target output device in real-time. The standard error stream, for example, is normally unbuffered. This is done so that any error messages are displayed as quickly as possible, regardless of whether they contain a newline or not. *Write* generally behaves this way -- or at least for this project we can think of it as being this way.

Based on the material presented in lecture, you should also realize that processes are selected from the ready queue as dictated by some scheduling algorithm. Processes are executed in the order determined, not by the user, but by the scheduler. If a single program spawns several concurrently active processes, the exact execution order of each process can be different (depending on system load, or even by sheer chance) for the program on different invocations of the exact same program code.

Normally, process scheduling is transparent to the end user. The rules of output buffering are also designed to maximize system performance while being transparent to the end user. However, when the randomness of process scheduling is combined with the "delayed" output effects of buffering, idiosyncrasies may arise and become apparent to the end user in the form of missing, out-of-order, or extraneous output. These anomalous results are caused by some interesting race conditions that occur when multiple processes, each of which performs some printing (and each of whose output is being buffered), are actively running concurrently. For example, when a parent process terminates, all of its children are terminated too. Typically, buffers get emptied upon process termination; however, if several parent/child processes are running, it is possible that the parent process will terminate and orphan the child process before its output can be flushed. The end user, in effect, experiences a situation where the child's output is lost. Other execution scenarios can actually lead to output being replicated.

Fortunately, there are ways for a programmer to counter the effects of buffered I/O and the seemingly randomness of scheduling. To obtain a more deterministic and consistent execution order for processes, the *wait* system call can be used to force a process to "pass up its turn" on the CPU until another process has a chance to catch up. Also, output can be forced to occur, i.e., flushed by the user via the *fflush(stdout)* statement. In these exercises, the *wait* statement will be used to force parents to wait (if necessary) for the child process to complete. The utility of the *fflush* system call is also demonstrated.

**1)** Execute the following 'C' program, first interactively, then by redirecting output to a file at the UNIX shell level with a ">". Explain the difference between the output observed on the terminal and that contained in the target piped file. [2 pts]

```
int main (void)
{       printf("Line 1 ..\n");
        write(1,"Line 2 ",7);     }       Be sure there are 7 characters in "Line 2 "
```

**2)** Execute the following 'C' program, first interactively, then by redirecting output to a file at the UNIX shell level with a ">". Explain what has happened with the addition of the *fflush* system call. [2 pts]

```
#include <stdio.h>
int main (void)
{       printf("Line 1 ..\n");
        fflush(stdout);
        write(1,"Line 2 ",7);     }       Be sure there are 7 characters in "Line 2 "
```

**3)** Run the following 'C' program several times interactively. Note the different execution order on different runs.

```
main ()
{       int pid;
        int i;
        for (i=0; i<3; i++){
                if ((pid=fork()) <0 )    {printf("Sorry...cannot fork\n"); }
                else if (pid ==0)        {printf("child %d\n", i);        }
                else                     {printf ("parent %d\n", i);      }
                }
exit(0); }
```

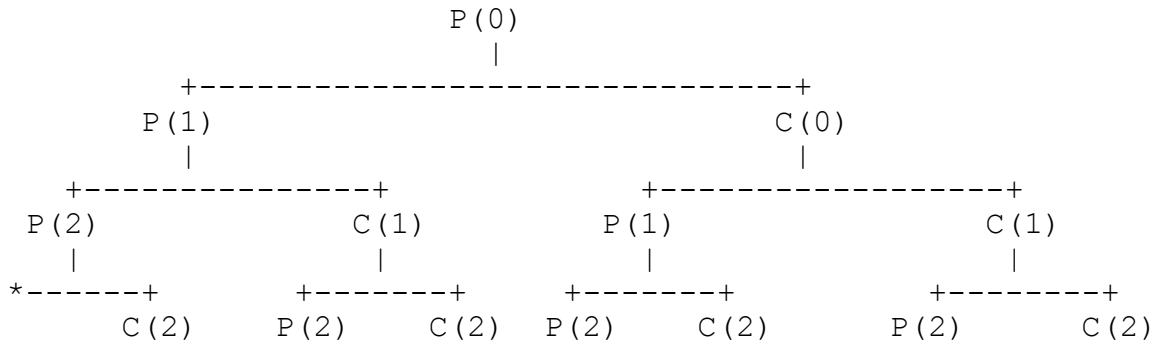
To help you understand the intended behavior of this program, note the following explanation of the **fork** system call along with the important points regarding this particular program's process execution tree. The system call **fork( )** is called without any arguments and returns an integer process identification number (pid). It causes the OS kernel to create a new process which is an exact duplicate of the calling process. The new process is termed to be a child of the parent process. The new child process is an exact clone of the parent. It has the same data and variable values as the parent at the time fork was executed. It even shares the same file descriptors as the parent. The child process does not start its execution from the first instruction in the source code, but continues with the next statement after the call to fork. That is, after the call, the parent process and its newly created offspring execute concurrently with both processes resuming execution at the statement immediately after the call to fork.

This leads to an intriguing predicament because if the parent and child are perfect clones, how does the child know it is a child and the parent know it is the parent ? The only way to tell is to have each process immediately examine the return value of the fork call. In the parent, a successful fork returns the process identifier (PID) of the new child. The pid is set to a unique, non-zero, positive integer identifying the newly created child process for the parent. In the child, fork returns a nominal value of 0. The value of the pid enables a programmer to distinguish a child from its parent and to specify different actions for the two processes, usually via an IF or CASE statement. A process can obtain its own pid and that of its parent using the **getpid( )** and **getppid( )** system calls respectively. The typical method of spawning processes is as follows. The main (parent) program executes a fork. If the fork is successful, each process must now determine its identity (parent or child) by checking the value returned by fork. Then, a branch in execution paths occurs as a function of the process type (parent or child) through a simple test of the return value of the fork system call.

The behavior of the fork may seem a little counterintuitive if you are being introduced to it for the first time. The key to understanding it is to think in terms of processes instead of programs. Normally, when you produce a program, you think of each line of the source code text as being executed in a predictable sequence. Typically, when you run the program, you run a single process on your source code. However, when you think of processes, you have to think of each instance of your program behaving as an independent entity. Each process may share the same program source code, but after forking, each process may pursue a completely different route through the program. Also bear in mind that, on a uniprocessor implementing multiprogramming, there is only one CPU. Therefore, only one process is really executing at any point in time, even though from a programmer perspective, they are running "concurrently". Depending on the scheduling algorithm employed by the system, the parent and child processes can make progress at different rates of execution.

In particular, for program 3, note the following:

* A process P in an iteration will continue and try to iterate with a value of i incremented by 1. P will have generated a child C that will also try to iterate with a value of i incremented by 1. We can represent the various processes with the tree:




* In this tree, each node is represented with the value of i at the time this process prints a message.

* Also, in this tree, when we go from a node N to its left successor, we go from one iteration to the next iteration of the process represented at N.

* When we go from a node N to its right successor, we are introducing the child of the process represented at N.

In what order are the 'nodes' of the process tree traversed ? That is, left or right most, depth or breadth first and why ? [3 pts]

**4)** Making the minor changes to program 3 above needed to get the code below, execute the following 'C' program several times interactively.

**a)** Explain how and why the order of the output from this program is different from that of program 3. [3 pts]

```
main ()
{       int pid;
        int i;
        for (i=0; i<3; i++){
                if ((pid=fork()) <0 )        {printf("Sorry...cannot fork\n");     }
                else if (pid ==0)            {printf("child %d\n", i);             }
                else                         {wait();
                                                printf ("parent %d\n", i);        }
                }
exit(0); }
```

**4b)** Run the program several times while redirecting output to a file via ">". First, note that the standard output is line buffered if it's connected to a terminal device, otherwise it's fully buffered. When we run the program interactively, we get only a single copy of the *printf* lines because the standard output buffer is flushed by the newline. However, when we redirect standard output to a file, we get multiple copies of some of the *printf*s. What has happened in this case of full buffering is that a printf before a fork is called once, but the line "printed" remains in the buffer when fork is called. This buffer is then replicated and inherited by the child process. Both the parent and child now have a standard I/O buffer with the "printed" line in it. Any additional *printfs* performed by the parent or child simply appends additional printed data to its (now separate) existing buffer. When each process terminates, its copy of the buffer is finally flushed.

**5)** Making the minor changes to program 4, execute the following 'C' program several times interactively, as well as several times while redirecting to a file. Explain what has happened. [3 pts]

```
#include <stdio.h>
main () {
        int pid;
        int i;
        for (i=0; i<3; i++){
                if ((pid=fork()) <0 )      {printf("Sorry...cannot fork\n"); }
                else if (pid ==0)              {printf("child %d\n", i);
                                                    fflush(stdout);                    }
                else                           {wait();
                                                    printf ("parent %d\n", i);
                                                    fflush(stdout);                    }
                }
exit(0);}
```

*For the problems 1-5 above, provide a run-time trace (or a simple "print screen" snapshot) of the execution result(s) along with a written explanation documenting your observations of what the program did and why. Your written explanation carries the heaviest weight in the evaluation of your answers for problems 1-5. Simply providing printouts from the program is not sufficient. In fact, it is not necessary to provide a printout of the source code from problems 1-5 because it is already given to you.*

*Note that you do not have to provide source code for problems 1-5; however, you do need to provide the source code for problems 6-8.*

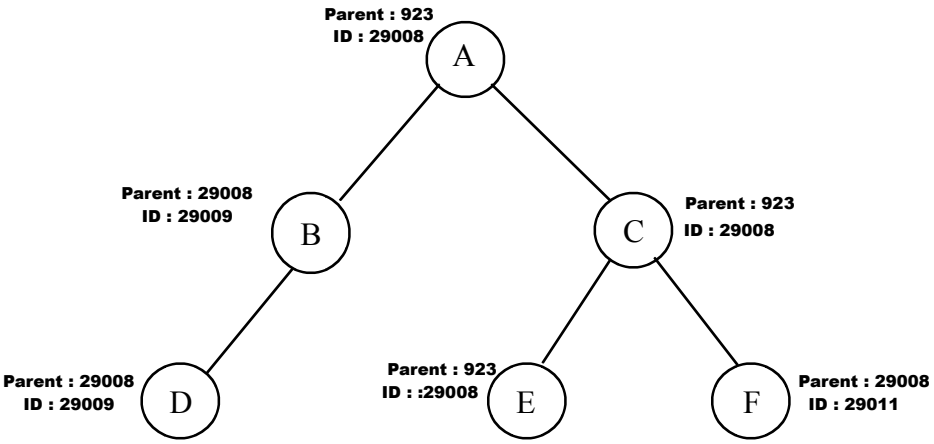
*Now that the preliminaries have been discussed, the student is ready to write his/her own programs and experiment with forking. In this exercise, the focus is on OS support of user process management tasks. Specifically, in a multiprogrammed system where several processes can be active concurrently, functions are needed to enable user processes to: 1) create (fork) other processes, 2) coordinate (via wait and sleep) their execution sequences, and 3) communicate with (signal) each other.*

*For the following programs: Hand in your source code listing and a printout showing that the program performs correctly for various test conditions. Mark and label your printout using a pen/pencil/highlighter to identify the output of your program for given inputs. Note that your documentation of how well your program performs under various test cases will be part of the evaluation criteria. Simply handing in source code listings will not earn the student full credit. Although some information is provided below regarding the various system calls, you may find it useful to consult other reference documentation such as that contained on-line or in other books. Note that you may also need to make use of the fflush and wait system calls as demonstrated in preliminary exercises 1 to 5 in order to make the redirected output appear correctly.*

**6)** Write a program that will create a child process. Have the parent print out its pid and that of its child. Have the child print its pid and that of its parent. Have the processes print informational messages during various phases of their execution as a means of tracing them. A typical printout might contain the following output (not necessarily in this order). [10 pts]

Immediately before the fork. Only one process at this point.
Immediately after the fork. This statement should print twice.
Immediately after the fork. This statement should print twice.
I'm the child. My pid is XXXX. My parent's pid is XXXX.
I'm the parent. My pid is XXXX. My child's pid is XXXX.

**7)** Write a program that will create a process tree structure as shown below. Again, have the processes print informational messages to verify that their parent-child relationship is that as shown. So processes B and C should both report the same parent pid (that of A). Also, processes E, and F should both report the same parent pid (that of C) and D should report its parent as being B. Label each node in the figure with the PID of the process your program creates. [30 pts]




**8)** One way for a parent process to attack a very large problem might be to split it into several smaller pieces, create several new child processes, and allocate each child a piece of the problem. In this and other scenarios, it is important that processes be able to synchronize with each other. The **wait(&status)** function provides one mechanism in which two processes can re-synchronize at some point in their executions. It causes a parent process to be suspended until some child process terminates. In some ways, it is a specialized version of the **sleep(x)** function which causes a process to suspend itself for x seconds.

Write a program that will create a child process. Have the child sleep for 5 seconds; have the parent wait for the child to finish sleeping. Put print messages in the program such that you can keep track of where each process is. For example, the following strings would enable you to compare the time-based execution with and without the parent waiting. [10 pts]

Child going to sleep.
Parent starting wait.
Child finished sleeping.
Parent finished wait.

**9)** You have access to CPU's capable of performing at 1 MIP each. Your goal is to design a new system that has 2 MIPs of performance. You can either redesign 1 of the current processors such that it is twice as fast, or use two of the existing processors in a parallel processing fashion. Which approach would generally give a better cost/performance ratio? Use a simple diagram to defend your answer. [3 pts]

**PDF s39**

**10)** One of the design goals of Instruction-Level Parallelism (ILP) was to make it transparent to the software layer; however, there are limitations as to how effective it is, and to what degree of aggressiveness or "look-ahead" it can be performed at. What are the causes of the limitations and how do they arise? [3 pts] **Day 7, pg51**

**11)** What is HyperThreading, and what, if anything, should software programmers do to leverage it as effectively as possible. [3 pts]

**pg 78**

**12)** What are the micro-architecture and macro-architecture trends and HW design problems that have led to the need for programmers to explicitly express parallelism into their software ? What are some of the challenges associated with directly exposing the underlying hardware parallelism to the software layer and programmers ? [3 pts]

**initial Paras**

**13)** Describe the four Flynn Architecture Classifications. Explain why the MISD configuration has traditionally thought of as non-sensible. [3 pts]

**14)** In some ways, the speedup from pipelining is "an illusion" because each instruction may actually take longer to complete in a pipelined machine as compared to a non-pipelined one. Explain how and why this statement is true. [3 pts] **pg 30**

**b)** Under extremely poor conditions, the pipeline could conceivably be stalled and flushed after every single instruction. In this situation, how would the performance of the pipelined machine compare against that of the non-pipelined one? Why? [2 pts]

**15)** Assuming that an instruction's execution is broken into four equal stages (FI, DA, FO, and EX), and that the pipeline is simply stalled and flushed in order to accommodate worst-case analysis on branches, show the pipeline schedule for the instruction stream below. [5 pts]

```
S1:   X = 5
S2:   Y = X + X
S3:   IF Y > 7 THEN GOTO S5
S4:   A = B + C
S5:   Z = X + Y
```

| | T1 | T2 | T3 | T4 | T5 | T6 | T7 | T8 | T9 | T10 | T11 | T12 | T13 | T14 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **S1** | | | | | | | | | | | | | | |
| **S2** | | | | | | | | | | | | | | |
| **S3** | | | | | | | | | | | | | | |
| **S4** | | | | | | | | | | | | | | |
| **S5** | | | | | | | | | | | | | | |

**16)** Assume a simple single instruction lookahead issuing scheme for instruction-level parallel processing while preserving apparent sequentiality in which the control unit issues consecutive instructions until a hazard is detected. At that point, all issuing stops until the blocked statement can execute. Show the instruction schedule for the stream S1 to S7 for the two multi-function hardware configurations below. Also, show the statement number and type of hazard on which issuing is blocked for each time segment.

S1: A = B + C
S2: D = E + F
S3: G = A * Y
S4: Z = H + G
S5: W = I * J
S6: F = W * Z
S7: H = K * L

**a) CASE 1: Two adders and one multiplier unit available. [2 pts]**

| *Time* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Adder 1 | | | | | |
| Adder 2 | | | | | |
| Multiplier | | | | | |
| *Hazard:* | | | | | |

**b) CASE 2: Two adders and two multiplier units available [2 pts]**

| *Time* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Adder 1 | | | | | |
| Adder 2 | | | | | |
| Multiplier 1 | | | | | |
| Multiplier 2 | | | | | |
| *Hazard:* | | | | | |

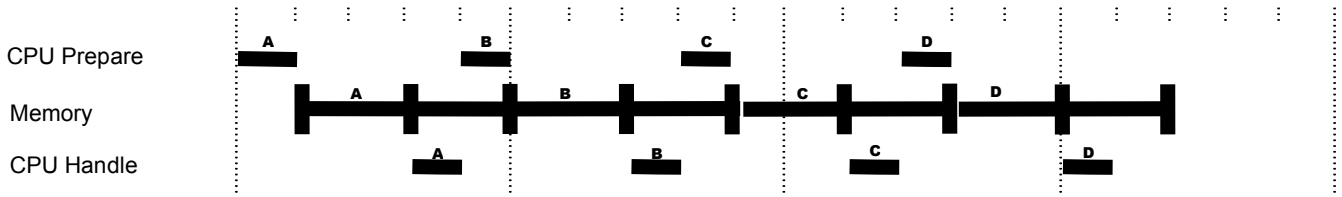
**c) CASE 3: Four adders and four multiplier units available [2 pts]**

| *Time* | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| Adder 1 | | | | | |
| Adder 2 | | | | | |
| Adder 3 | | | | | |
| Adder 4 | | | | | |
| Multiplier 1 | | | | | |
| Multiplier 2 | | | | | |
| Multiplier 3 | | | | | |
| Multiplier 4 | | | | | |
| *Hazard:* | | | | | |

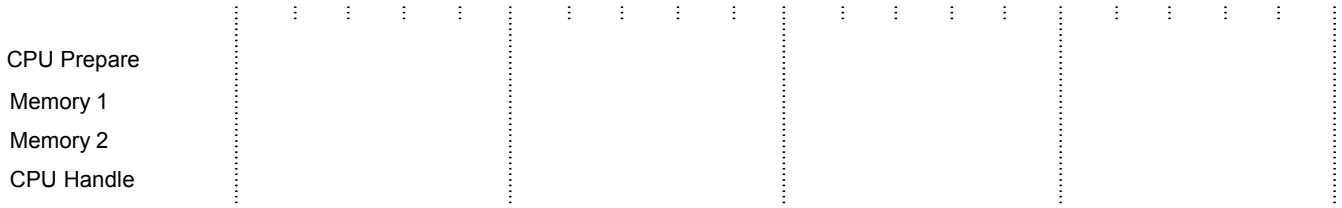
**17)** Assume that a RAM memory chip has a 0.4 ms access time and 0.8 ms cycle time. It takes the CPU 0.2 ms to prepare a memory request and 0.2 ms to process the result. The CPU can either prepare a memory request or process a result in parallel with the memory; however, it cannot prepare a request and process a result concurrently. If an idle memory chip is available, the CPU can issue a request for the next operand (e.g. B) to it before actually receiving and processing a previously requested one (e.g. A) from another memory chip, but it must handle a result as soon as it becomes available from memory.

A worksheet is provided for each of the various degrees of interleaving showing time from 0 ms to 4 ms in 0.2 ms increments. Assume a string of four operands (A - D). Show the time at which each operand is prepared, memory accessed (including wait times), and handled.

**a) Non-Interleaved [2 pts]**




**b) Two-Way Interleaved [2 pts]**




**c) Four-Way Interleaved [2 pts]**