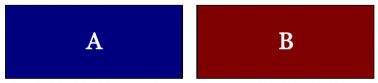
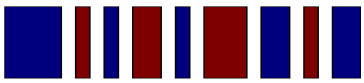
CSCI3240 Lab 10:
Process-based Concurrent Programming

Computer programming in which, during a period of time, multiple processes are being executed. Commonly we can say that two processes can be interleaved so that they are executed in turns.

For instance, imagine tasks A and B. One way to execute them is sequentially, meaning doing all steps for A, then all for B.



Concurrent execution, on the other hand, alternates doing a little of each task until both are all complete.



Concurrency allows a program to make progress even when certain parts are blocked. For instance, when one task is waiting for user input, the system can switch to another task and do calculations.

Fork System Calls:

The fork system call is a fundamental function in Unix-like operating systems that creates a new process by duplicating an existing process. The new process is called the child process, and the original process is called the parent process. The fork system call allows you to create a new process that is a copy of the parent process and then modify the child process to perform a different task.

The fork system call has the following prototype:

```
#include <unistd.h>

pid_t fork(void);
```

It takes no arguments and returns a process ID.

- ***Negative Value***: creation of a child process was unsuccessful.
- ***Zero***: Returned to the newly created child process.
- ***Positive value***: Returned to parent or caller. The value contains process ID of newly created child process.

When the fork system call is executed, the operating system creates a copy of the parent process, including its memory, variables, and file descriptors. The child process starts executing immediately after the fork system call, but it has its own copy of the parent process's memory and variables. This means that any changes made to the variables in the child process do not affect the variables in the parent process, and vice versa.

Here is an example program that demonstrates how to use the fork system call:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
    pid_t pid;

    pid = fork();

    if (pid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }
    else if (pid == 0) {
        /* Child process */
        printf("Hello from the child process!\n");
        exit(EXIT_SUCCESS);
    }
    else {
        /* Parent process */
        printf("Hello from the parent process!\n");
        exit(EXIT_SUCCESS);
    }
```

In this program, the fork system call is used to create a child process that prints a message to the console. The parent process also prints a message to the console and terminates. The order of the print message is not guaranteed.

When you compile and run this program, you should see the following output:

```
Hello from the parent process!
Hello from the child process!

or

Hello from the child process!
Hello from the parent process!
```

The order of the output may vary, depending on the scheduling of the processes by the operating system.

Example 2:

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

int main() {
  pid_t pid;
  int i;

  pid = fork();

  if (pid == 0){
      for (i = 0; i < 8; i++){
          printf("-child-\n");
      }
      return 0;
  }

  for (i = 0; i < 8; i++){
      printf("+parent+\n");
  }

  return 0;
}
```

Try to compile and run to see the output.

Exercise:

Try adding sleep function before the “-child-\n” print statement.

```
for (i = 0; i < 8; i++){
    sleep (rand()%4);
    printf("-child-\n");
}
```

This makes the program sleep for random number of seconds between 0 to 3 sec. Analyze the difference between two outputs.

Wait System Calls:

The wait system call in Unix-like operating systems is used to wait for the termination of a child process. When a process creates a child process using the fork system call, it can use the wait system call to wait for the child process to terminate and retrieve its exit status.

The wait system call has the following prototype:

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

pid_t wait(int *status);
```

The wait system call takes a pointer to an integer variable “status” as its argument. This variable will be used to store the exit status of the child process when it terminates. If the child process terminated successfully, the exit status will be 0. If the child process terminated due to an error, the exit status will be a non-zero value that indicates the type of error.

The wait system call returns the process ID (PID) of the child process that terminated, or -1 if there are no child processes to wait for.

Example 3:

```c
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/wait.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
   printf("I am: %d\n", (int) getpid());

   pid_t pid = fork();
   printf("fork returned: %d\n", (int) pid);

   if (pid < 0) { /* error occurred */
    perror("Fork failed");
   }
   if (pid == 0) { /* child process */
    printf("I am the child with pid %d\n", (int) getpid());
               printf("Child process is exiting\n");
               exit(0);
       }
   /* parent process */
   printf("I am the parent waiting for the child process to end\n");
       wait(NULL);
       printf("parent process is exiting\n");
       return(0);
}
```

Exercise:

Assume the parent process id is 2337 and child process id is 2338. Predict the output of the above example 3.

Programming practice questions:

1. Write a program that uses the fork system call to create a child process that prints "Hello, world!" to the console. The parent process should wait for the child process to finish before terminating.

2. Write a program that uses the fork system call to create two child processes that each print a message to the console. The parent process should wait for both child processes to finish before terminating.

3. Write a program that uses the fork system call to create a child process that reads an integer from the console and calculates its factorial. The child process should then print the result to the console, and the parent process should wait for the child process to finish before terminating.

4. Write a program that uses the fork system call to create a child process that reads a string from the console and converts it to uppercase. The child process should then print the uppercase string to the console, and the parent process should wait for the child process to finish before terminating.

5. Write a program that uses the fork system call to create a child process that reads a list of integers from the console and finds the sum of the even numbers. The child process should then print the result to the console, and the parent process should wait for the child process to finish before terminating.