# E0-253: Operating Systems
# Major Programming Assignment

**Background**

The C programming language is unsafe by design. In a nutshell, C does not check the validity of memory accesses—arbitrary values (e.g., integers) can be treated as pointers. As a result, any part of application can manipulate arbitrary locations (e.g., program heap). It is therefore up to the programmers to reason about the correctness of their code in the absence of any language-level protection.

In this assignment, you are required to build an operating system abstraction (a system call) for user-space programs to save/restore their program state in memory. An application can invoke this system call to save the program state before executing potentially unsafe code and to restore the saved program state after the unsafe code has finished.

**A simple solution**

One way to achieve the aforementioned goal is via `fork`. This approach creates a child process using `fork` and delegates unsafe code to the child. Since modifications performed by a child process are not visible to the parent, we can restore a safe program state by resuming in the parent after child has finished executing unsafe code. However, this approach is not very efficient due to the overhead of creating an additional process and switching between parent and child.

**Assignment**

In this assignment, you will introduce a `context`-based method as an alternative to the `fork`-based method in Linux to achieve the aforementioned goal. The primary difference between the two approaches is that `fork` relies on an additional process to execute the unsafe code while the `context`-based method will save/restore program state without creating an additional process.

You will extend your `my_precious` system call in Linux to enable fast operating system support for saving and restoring program state. The system call will accept one integer argument (either 0 or 1) and will be responsible for:

1. Saving the program state (if the argument is 0). Any user-space program can invoke the system call to save its program state before executing potentially unsafe code.

2. Restoring the program state (if the argument is 1). This will restore memory content to the same state as it were just before saving the context.

**Details and assumptions**

1. A program state typically refers to a combination of many things: stack, heap, files, signals and registers. As part of this assignment, we are concerned only about the state of dynamically allocated anonymous memory (e.g., memory allocated via `brk, sbrk, mmap` system calls)—you can ignore the rest of the program state in your design. You can also ignore "process stack".

2. Return 0 from the system call if save/restore is successful. Otherwise, return an appropriate error code.

3. Maximum one context can be saved at any point. If the system call is invoked to save the program state with an already existing context, return -EINVAL. Similarly, if it is invoked to restore without saving a context, return -EINVAL.

4. Only single-threaded programs will be considered as part of this assignment.

5. Assume that address space will not be manipulated between the points of saving and restoring the context (i.e., no calls to `malloc/mmap, munmap/free`). Your system call is also expected to NOT manipulate process address space (i.e., no vm regions should be added/removed).

6. Make sure to clean up process state upon termination. A program may terminate after saving the context but before restoring it. This should not lead to memory leak.

**Evaluation**
Evaluation will be based on two key aspects:

1. **Functional correctness:** Implementation should work for arbitrary program sizes.

2. **Performance:** Your implementation will be compared against the `fork` based method. Performance will be measured in terms of throughput (i.e., how many times `fork` and `my_precious` system calls can be executed per second, for different program sizes). Higher is better.

**Deliverables and timeline:**

1. Kernel patch for saving context (**Due date: Feb 25**)

2. Kernel patch for restoring the saved context (**Due date: March 25**)

3. Report with a brief description of your design, implementation and performance comparison with the `fork`-based method. Also discuss some of the major limitations of your implementation and how you can optimize it further (**Due date: March 31**)

4. Extending your system call to support THP (Transparent Huge Pages). Extend your report to include performance evaluation with THP (**Due date: April 15**)

**Test-cases**
Clone the following git repository and follow the README.md for instructions on how to compile and run.
`https://github.com/apanwariisc/e0253-os-2020.git`

**Submission Instructions:**
TBD.

**References:**
[1] Understanding the Linux Kernel (3rd Edition) by Daniel P. Bovet and Marco Cesati (Chapter 3 and 9).
[2] Understanding the Linux Virtual Memory Manager by Mel Gorman (Chapter 3, 4 and Appendix D in Code Commentary).
[3] Look at the following files in Linux source to get started:
`include/linux/mm_types.h`
`include/linux/mm.h`
`mm/pagewalk.c`
`mm/memory.c`.