CENG328
Operating Systems

Laboratory X
Memory Management

1.1 Memory Management

- Memory management is the art and the process of coordinating and controlling the use of memory in a computer system. Memory management can be divided into three areas:
  - ***Memory management hardware (MMUs, RAM, etc.)*** consists of the electronic devices and associated circuitry that store the state of a computer. These devices include RAM, MMUs (memory management units), caches, disks, and processor registers. The design of memory hardware is critical to the performance of modern computer systems. In fact, memory bandwidth is perhaps the main limiting factor on system performance.
  - ***Operating system memory management (virtual memory, protection)*** is concerned with using the memory management hardware to manage the resources of the storage hierarchy and allocating them to the various activities running on a computer. The most signicant part of this on many systems is virtual memory, which creates the illusion that every process has more memory than is actually available. OS memory management is also concerned with memory protection and security, which help to maintain the integrity of the operating system against accidental damage or deliberate attack. It also protects user programs from errors in other programs.
  - ***Application memory management (allocation, deallocation, garbage collection)*** involves obtaining memory from the operating system, and managing its use by an application program. Application programs have dynamically changing storage requirements. The application memory manager must cope with this while minimizing the total CPU overhead, interactive pause times, and the total memory used.

1.1 Memory Management

- The **malloc()** function allocates an uninitialized memory block. It allocates a specied number of bytes of memory, as shown in the following prototype, returning a pointer to the newly allocated memory or NULL on failure:

```
void *malloc(size_t size);
```

- The **calloc()** function allocates and initializes a memory block. The difference is that **calloc()** initializes the allocated memory, setting each bit to 0, returning a pointer to the memory or NULL on failure. It uses the following prototype:

```
void *calloc(size_t nmemb, size_t size);
```

- The **realloc()** function resizes a previously allocated memory block. Use **realloc()** to resize memory previously obtained with a **malloc()** or **calloc()** call. This function uses the following prototype:

```
void *realloc(void *ptr, size_t size);
```

  - The ptr argument must be a pointer returned by **malloc()** or **calloc()**.
  - The size argument may be larger or smaller than the size of the original pointer.

1.1 Memory Management

- The **free()** function frees a block of memory. This function uses the following prototype:

```
void free(void *ptr);
```

  - The ptr argument must be a pointer returned from a previous call to malloc() or calloc().
  - It is an error to attempt to access memory that has been freed.

  Memory allocation functions obtain memory from a storage pool known as the *heap*.

- The **alloca()** function allocates an uninitialized block of memory. This function uses the following prototype:

```
void *alloca(size_t size);
```

  alloca() obtains memory from the process's stack rather than the heap and, when the function that invoked **alloca()** returns, the allocated memory is automatically freed.

1.2 Examples & Exercises

- **Using Dynamic Memory Management Functions;** code45.c
  - Illustrates the standard library's memory management functions.
  - Lines 15-18 illustrate **malloc()** usage. It is attempted to allocate ten bytes of memory, check **malloc()**'s return value, display the contents of the uninitialized memory, and then return the memory to the *heap*.
  - Lines 20-22 repeat this procedure for **calloc()**.
  - Rather than freeing d, however, it is attempted to extend it on lines 26-28. Whether **realloc()** succeeds or fails, it should still point to the string "foobar".
  - The pointer, e, as shown on lines 31-33, is allocated o the stack and, when **main()** returns (that is, when the program exits), its memory is automatically freed.
- **Dangling Pointers;**

```
char *str;
str = malloc(sizeof(char) * 4)
free(str);
strcpy(str, "abc");
```

  - Write a program that contains the code segment above.
  - What kind of problem do you expect to come up with? Why?

1.2 Examples & Exercises

- **A Problem Child;** code46.c.
  - C assumes you know what you are doing, most C compilers ignore uses of uninitialized memory, buffer overruns, and buffer underruns.
  - Nor do most compilers catch memory leaks or dangling pointers.
  - Bugs in the program:
    - A memory leak (line 18),
    - Overruns the end of dynamically allocated heap memory (lines 22 and 28),
    - Underruns a memory buffer (line 32),
    - Frees the same buffer twice (lines 36 and 37),
    - Accesses freed memory (lines 40 and 41),
    - Clobbers statically allocated stack and global memory (lines 48 and 44, respectively).
  - These bugs can prevent the program from executing depending on the configuration (to allow core dumps), but leaks and clobbered memory usually show up as unpredictable behavior elsewhere in the program.
  - Fix the errors (if occurs).

2. Laboratory Review

- **fork()** and **exec()**,
  - Write a program that contains the following lines of code.
    - Include necessary header files (tip: use man pages),
    - Fill in the blanks,
    - When compiled; no errors or warnings should be encountered,
    - When execute; a newly created child process should execute "echo Hello again, exec!", using **execl** function.

```
int status;
pid_t child_pid;
printf("Parent process: creating a child process to execute echo command.\n");
if ((..........) == -1)
{
    printf("Can't create child process, terminating.\n");
    exit(EXIT_FAILURE);
}
else if (..........)
{
    printf("Child process: executing echo:\n");
    execl(.........., NULL);
}
else
{
    wait(&status);
    printf("Parent process: exiting.\n");
}
```

2. Laboratory Review

- **fork()** and **named pipes**,
  - Write a program that contains the following lines of code.
    - Create a fifo file (/tmp/fifo) from a terminal window first,
    - Include necessary header files (tip: use man pages),
    - Fill in the blanks,
    - When compiled; no errors or warnings should be encountered,
    - When executed; parent process should send a message to child process using a named pipe (FIFO).

```
char fifo_path[] = "/tmp/fifo", data[] = "Hello again, IPC!";
int status, data_length = 1 + strlen(data);
char *buffer = (char*)calloc(.........., ..........);
pid_t child_pid;
if ((child_pid = fork()) == -1) {
    printf("Can not create child process, terminating.\n");
    exit(EXIT_FAILURE);
} else if (child_pid == 0) {
    FILE* fifo = fopen(..........);
    fgets(..........);
    fclose (fifo);
    printf("Child process:  received message \"%s\"\n", buffer);
} else {
    printf("Parent process: sending message  \"%s\"\n", data);
    FILE* fd = fopen(..........);
    fputs(..........);
    fclose (fd);
    wait(&status);
}
```