Advance Systems Programming

Lecture notes from **Prof. PC Ranga** *Scribe:* ***0xskaper (Rajat Yadav)***

Lecture: 0

1. **Kernel**
   - It is the core of the unix OS.
   - It directly interact with the hardware and provide service to the applications.
   - **System calls** are used to interact with kernel in order to request service.
   - **Libraries** of common functions are built on top of the systems call interface, but application are free to use both.
   - **Shell** is a special application that provides an interface for running other applications.
   - ***Note:*** *No application can directly interact with the kernel. (NO BYPASSING THE SYSTEM CALLS)*
     - **Linux** is the kernel for the GNU OS.
2. **Shells**
   - It is a command-line interpreter that reads user input and executes commands.
   - The user interact with shell is normally via a **terminal** or somtimes from a file called ***shell script***.
   - **COMMON SHELLS:**
     1. Bourne shell - **/bin/sh**
     2. Bourne-again shell - **/bin/bash**
     3. C shell - **/bin/csh**
     4. Korn shell - **/bin/ksh**
     5. TENEX C shell - **/bin/tcsh**
     6. Z shell - **/bin/zsh**
   - **COMMON COMMANDS AND VARIABLES**
     1. **$SHELL** - hold the name of current shell.
     2. **uname -a** - gives you the linux version.
3. **File System**

- Unix file system is a hierarchical arrangement of directories and files.
- Everything starts from the directory called **root**, whose name is **‘/’**.
- **Unix** treats everything as a file, even the directories are files which contains directory entries i.e **filename, structure information about attributes of files and other directories.**
- **Attributes** of a file:
  1. File Name
  2. Size
  3. File Type (File or Directory)
  4. Owner
  5. Permissions
  6. Last Modification
- []**stat()** in C returns all the information containing attributes of the file.
- **Home directory** is denoted by ~ symbol.

4. **Filename**
   - The only 2 characters that are forbidden in a filename are the ‘/’ and the NULL character.
   - Filenames are **Case-Sensitive.**
   - **/** - seprates filenames that form a pathname.
   - **NULL** - terminates a pathname.
   - **POSIX** recommends restricting filenames to consist of the following characters: letter (a-z, A-Z), numbers (0-9), period (.), dash (-), and underscore _ .
   - **. (dot) and .. (dot-dot)** automatically get created in every directory.
   - **. (dot)** - refers to the current working directory.
   - **.. (dot-dot)** - refers to the parent of the current working directorty.

5. **Pathname**

- A sequence of one or more filename seprated by slashed and optionally starting with a slash forms a pathname. **(e.g /mimir/AdvanceSystems)**
- **Absolute pathname** - begins with a '/' **(eg. /home/bin/zsh)**
- **Relative pathname** - file relative to the current working directory. **(eg. ../mimir/AdvanceSystems)**
- **/** - is special case for absolute pathname that has no file component.

6. **Input and Output** (IO)
    - 3 Files - **Standard Input, Standard Output and Standard Error**
    - All shell open 3 file descriptor whenever a new routine or program runs.
        1. **Standard Input:** File descriptor No. 0
        2. **Standard Output:** File descriptor No. 1
        3. **Standard Error:** File descriptor No. 2
    - Most shell provide a way to redirect any or all of these 3 descriptors to any file. **(eg. `ls > listing.txt`)**
    - **>** : output redirection
    - **<** : input redirection
    - **>>** : output concatenation

```
$ ls -l > output.txt # output of ls is stored in output.txt
$ sort < output.txt # content of output.txt goes into sort routine
and gets sorted.
$ echo $SHELL >> output.txt # append current shell to end of the
output.txt file.
```

1. **Pipes**
    - Its is a way to redirect output of a program to a input of another program.

    ```
    ls -l | grep .c | wc -w # ls gives all the file, then grep .c
    filter out all the C files and finally we get the word count.
    ```

2. **Processes and Process IDs**
    - an execution instance of a program is called a process.
    - Some OS call them *task*.

- UNIX system guarantees that every process has a unique numeric identifier called Process ID. The process ID is a non -ve integer.
- Every process has a process ID, parent process ID and group ID.
- ***NOTE***: **swapper** and **sched** has a process ID of 0.

Lecture: 1

1. **Introduction**
    - Pointers are crucial to C, and they are mainly used to:
        1. provide the way by which function can modify their calling arguments.
        2. Support dynamic allocation of memory.
        3. Refer to a large data structure in a simple manner.
        4. Support data structure like linked lists.
2. **Pointers are Addresses**
    - A pointer is a variable whose value is a memory addresses.
    - A pointer variable similar to other variables.
    - A pointer variable store address of another variable.
    - `base type *pointerName`
    - **'*'** - is used to dereference or declare a pointer.
    - **&** - is used to access memory location.
3. **NULL**
    - It used to indicate that pointer is not pointing to any valid data.
    - **NULL** is defined in **`stdlib.h`**
    - **NULL** is used to indicate a failure operation.
4. **Pointer Arithmetic**
    - When a pointer is incremented (or decremented). It will points to memory location of next element of its base type.
    - `ptr + 1 = 204 //ptr is int and int is 4 bytes`
5. **Pointers and Array**

- The name of the array is the address of the array’s first element.
- e.g `int arr[3];` reserves 3 consecutive unit of memory, each block of size 4.
- Pointers and array are interchangeable.
- **Limitations of Pointer arithmetic**
  1. *, / and % can’t be used with pointers.
  2. 2 pointers **cannot** be added together.
  3. Only **integer offsets** can be added to or subtracted from a pointer.
  4. A pointer p1 can be subtracted from p2. The result is an integer, the number of elements between p1 and p2.

6. **Generic Pointers**
   - A pointer variable defined as **void** is a generic pointer variable. (It can point to any type)
   - **Advantage:** not type casting is required.
   - You can’t dereference a generic pointer.
7. **Dynamic Memory Allocation**
   - **Static allocation:** a variable’s memory is allocated and persists throughout the entire life of the program. This is the case of global variable.
   - **Automatic allocation:** when a local variable are declared **inside a function**, the space for these variable is allocated when the function is called (starts) and is freed when the function terminates. This is also the case of parametric variables.
   - **Dynamic allocation:** allow a program at the execution time to allocate memory **when needed** and free it when no longer needed.
     1. **Advantage:** it is often impossible to know, prior to the execution time, the size of memory needed in many cases. (e.g. The size of an array based on any size n)
   - Most common used functions for managing dynamic memory are:

1. **void* malloc(int size):** to allocate a block (number of bytes) of memory of a given **size** and returns a pointer to this newly allocated block.
2. **void free(void *ptr):** to free previously allocated block of memory.

- ***CODE***

```
#include <stdio.h>

int main(void) {
  int num = 100;
  int *b;

  b = &num;
  printf("*b: %d\n", *b); // 100

  int **c;
  c = &b;
  printf("**c: %d\n", **c); // 100

  int ***d;
  d = &c;
  printf("***d: %d\n", ***d); // 100
}
```

```
#include <stdio.h>

int main() {
  int n1 = 10, n2;
  int *ptr;

  ptr = &n1;
  n2 = *ptr;
  printf("n1: %d  n2: %d  *ptr: %d\n", n1, n2,
         *ptr); // n1: 10  n2: 10  *ptr: 10
  return 0;
}

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

```c
int *createArray(size_t size) {
  int *ptr = malloc(sizeof(int));
  if (ptr == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return NULL;
  }
  return ptr;
}

int main() {
  int *arr = createArray(1000);
  if (arr == NULL) {
    return 1;
  }

  free(arr);
  return 0;
}
```

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

int *createArray(size_t size) {
  int *ptr = malloc(sizeof(int));
  if (ptr == NULL) {
    fprintf(stderr, "Memory allocation failed\n");
    return NULL;
  }
  return ptr;
}

int main() {
  int *arr = createArray(1000);
  if (arr == NULL) {
    return 1;
  }

  free(arr);
  return 0;
}
```

```c
#include <stdio.h>

int (*m1)(int, int);
int (*m2)(int);

int maximum(int a, int b) {

  if (a > b)
    return a;
  else
    return b;
}

int fact(int c) {
  int ret = 1;
  while (c >= 1) {
    ret = ret * (c--);
  }
  return ret;
}

int compute(int (*m1)(int, int), int (*m2)(int), int x1, int x2) {

  int ret1, ret2;

  ret1 = m1(x1, x2);
  ret2 = m2(ret1);

  return ret2;
}

int main(void) {

  int ret3 = compute(&maximum, &fact, 5, 3);

  printf("\nThe factorial of max(x1,x2) is %d\n", ret3);
}
```

Lecture 2

1. **File IOs**
   - File descriptors are used by kernel to address a open file. **(NON -VE)**
   - **1, 2 and 3** are reserved and are assigned to standard IO and error.

- UNIX has declared the reserved one with a symbolic constant, defined in `<unistd.h>` for these values are:
  1. **STDIN_FILENO:** Standart Input - 0
  2. **STDOUT_FILENO:** Standard Output - 1
  3. **STDERR_FILENO:** Standard Error - 2
- User processes can only get file descriptors > 2.
- File descriptor range from 0 through `OPEN_MAX` (a symbolic constant i.e defined by the OS).
- They are unbuffered cause they come under system calls.

2. **`open()`**
   - File is created and open by it. (Created only when file doesn’t alreadt exists)
   - **Function:**

   -
     ```
     #include<sys/types.h>
     #include<sys/stat.h>
     #include<fcntl.h>
     int open(const char *pathname, int oflag, mode_t mode);
     ```
   - **`pathname`:** absolute or relative.
   - **`oflag`:** formed by bitwise OR | constants. (e.g. O_RDONLY, O_WRONLY, etc)
   - **`mode_t`:** only used when creating file.
   - When creating a file, `mode_t` goes through a default umask (*file permisson = asked file permission - umask*). (e.g. 0777 - 0022 = 0755)
   - returns a file descriptor if OK, -1 on error.
   - Final file permission = NOT (umask) AND (asked file permission)

3. **`read()`**
   - Data is read from an open file.
   - **Function**

```
#include<unistd.h>
ssize_t read(int filedes, void *buff, size_t nbytes);
```

- **`filedes`:** file descriptor.
- **`*buff\`** : buffer to store output.
- **`size_t`** : size of the output.

- returns: number of bytes read, 0 if end of file, -1 on error.
- also read the UTF character.

1. **write()**
    - Data is written to an open file with write function.
    - **Function:**

```
#include<unistd.h>

ssize_t write(int filedes, const void *buff, size_t nbytes)
```

1. **close()**
    - free the file descriptor.
    - returns 0 when OK and -1 otherwise. (e.g. -1 if already closed and try to close again)
2. **lseek()**
    - It is used to change the file offset.
    - **Function:**

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

off_t lseek(int filedes, off_t offset, int whence);
```

- returns new file offset if OK, -1 on error.
- by default it is intialized to 0, when a file is opened, unless O_APPEND is specified.
- **SEEK_SET**: offset is set to *offset* bytes from the beginning of the file.
- **SEEK_CUR**: offset is set to current value + *offset*. (*offset* can be +ve and -ve)
- **SEEK_END**: offset is set to the size of the file plus the *offset*. (The *offset* can be +ve or -ve)
- ***NOTE***: UTF characters are included and have a size of 8 bits. (UTF character is can also be changed to n)

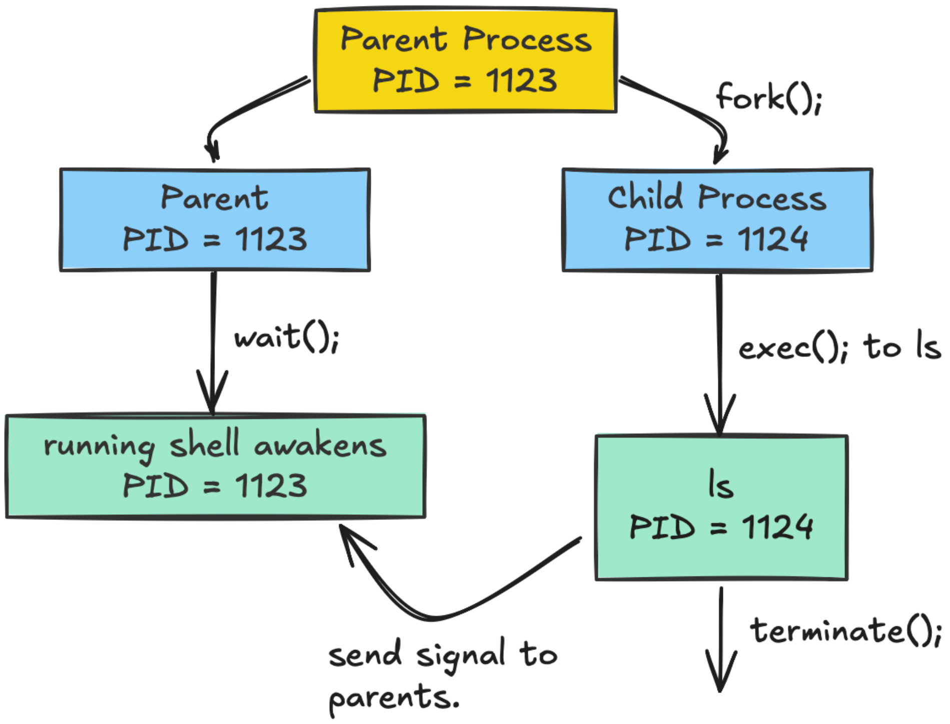
```
// ALL LECTURE 2 CODE
#include <fcntl.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
```

```
int main() {
  int fileDescriptor = open("check.txt", O_CREAT | O_RDWR, 0777);
  printf("File Descriptor: %d\n", fileDescriptor); // FD = 3

  char *writeBuff = "1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
19 20";
  long int byteWritten;

  byteWritten = write(fileDescriptor, writeBuff,
strlen(writeBuff));
  printf(
      "Buffer: %s\n",
      writeBuff); // Buffer: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20
  printf("Byte Written: %ld\n", byteWritten); // Byte Written: 50
  close(fileDescriptor);

  fileDescriptor = open("check.txt", O_CREAT | O_RDWR, 0777);

  char *buff1[100];
  long int byteRead;

  byteRead = read(fileDescriptor, buff1, 100);

  printf("Bytes read: %ld\n", byteRead); // Bytes read: 50
  printf("Buffer: %s\n",
         buff1); // Buffer: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16
17 18 19 20

  close(fileDescriptor);

  //
--------------------------------------------------------------------

  fileDescriptor = open("check.txt", O_CREAT | O_RDWR, 0777);
  char *buff2[100];
  int offset;
  offset = lseek(fileDescriptor, 10, SEEK_SET);
  char *writeBuff2 = "Bald Headed Demon";
  byteWritten = write(fileDescriptor, writeBuff2,
strlen(writeBuff2)); // 17
  printf("Offset: %d\n", offset);
  printf("Byte Written: %ld\n", byteWritten);
  // Offset: 10
  // Byte Written: 17
  // 1 2 3 4 5 Bald Headed Demon13 14 15 16 17 18 19 20

  offset = lseek(fileDescriptor, 10, SEEK_CUR); // SEEK_CUR = 27
  printf("Offset: %d\n", offset);
  char *writeBuff3 = "D";
  byteWritten = write(fileDescriptor, writeBuff3,
strlen(writeBuff3));
  printf("Byte Written: %ld\n", byteWritten);
  // Offset: 37
  // Byte Written: 1
  // 1 2 3 4 5 Bald Headed Demon13 14 15 1D 17 18 19 20

  offset = lseek(fileDescriptor, 0, SEEK_CUR);
  printf("Offset: %d\n", offset);
  // Offset: 38

  offset = lseek(fileDescriptor, -100, SEEK_CUR);
  printf("Offset: %d\n", offset);
  // Offset: -1 (offset don't move cause we can't go back 100 bytes
from 38)

  offset = lseek(fileDescriptor, 100, SEEK_END); // SEEK_END = 50
  printf("Offset: %d\n", offset);
  byteWritten = write(fileDescriptor, writeBuff3,
strlen(writeBuff3));
  printf("Byte Written: %ld\n", byteWritten);
  // Offset: 150
  // Byte Written: 1
  // 1 2 3 4 5 Bald Headed Demon13 14 15 1D 17 18 19 20<HOLE OF
100>D

  offset = lseek(fileDescriptor, 100, SEEK_END); // SEEK_END = 151
  printf("Offset: %d\n", offset);
  byteWritten = write(fileDescriptor, writeBuff3,
strlen(writeBuff3));
  printf("Byte Written: %ld\n", byteWritten);
  // Offset: 251
  // Byte Written: 1
  // 1 2 3 4 5 Bald Headed Demon13 14 15 1D 17 18 19 20<HOLE OF
100>D<HOLE OF 100>D

  offset = lseek(fileDescriptor, 0, SEEK_CUR);
  printf("Offset: %d\n", offset);
  // Offset: 252

  close(fileDescriptor);

  return 0;
}
```

Lecture 3

1. **Unix Process**
   - Unix is a multi-user and multitasking OS.
   - Users can run their programs concurrently and shared hardware resources.
   - An **user can run multiple programs** and task concurrently.
   - It appears that execution is done in parallel, however in reality OS switches between mutliple users and process rapidly in an interleaved manner.
   - **Unix Processes:**
     1. Any routine is a process.
     2. ex.c (Program) > ex (executable) > ./ex (process).
   - Every process in UNIX has:
     1. PID
     2. Some code
     3. Some data
     4. Stack
     5. env
   - Unix start as a single process, called **init**, and it has a PID 1.
   - The only way to create a new process in Unix, is to duplicate an existing one.
   - The process init is the ancestor of all subsequent processes.
   - Process **init** never dies.
   - The **spawning** of new process is done with 2 **system calls.**
     1. **`fork()`**: duplicated the called process.

2. **exex()**: replaces the called process by a new one.



1. **fork()**
    - It is used to create new child process.
    - **Function:**

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

pid_t fork(void);
```

- returns 0 in child, PID of child in parent, -1 on error.
- is called **once**, but returns **twice**.
- The reason for returning 0 in child because child can do `getppid();` to get parent PID;
- The reason for returning child PID in parent because child can have more `fork();` to create more child.

- File descriptors are inherited, therefore **parent** and **child** have same file descriptors.
- **fork();** performs in cocurrency.
- Execution resumes at the line after the `fork()` statement in both **parent** and **child.**
- **Parent** and **Child** have different scope that means same variable can different values in execution.
- **Usage:**
  1. A process wants to execute another program.
  2. A program needs to execute subtask. (e.g. Server/ sockets)
- $$\text{Total Process} = 2^n - 1$$

1. **exit()**
   - used to terminate a process.

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

void exit(int status);
```

- does not return a value.
- The status value is available to the parent process through the `wait();` system call.
- When invoked by a process:
  1. closes all the process's file descriptor.
  2. free the memory that is consumed by the code, data and stack.
  3. send a **SIGCHILD** signal to the parent (every process has a parent) and waits for the parent to accept the its return code.
     - SIGCHILD is signal that indicates a process stareted by the current process has terminated.
     - **NOTE:** SIGCHILD is a signal and not a return value.
- **Termination Condition**
  1. When the `exit()` system call is called within a process.
  2. When the process has executed all his statements and follows the natural course of termination.

3. When the process is terminated by a signal.
4. **NOTE**
    - In all above cases (Normal and Signalled termination), the child sends `SIGCHLD` and wait for the parent to accept its return code.

1. **wait()**
    - It allows the parent to wait for one of its children to terminate and too accept its child's termination code.

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

pid_t wait(int* status);
```

- when called `wait()` can:
    1. Block the called process, until one of it's children terminates and to accept its child's termination code.
    2. or returns the PID of the child process, if a child has terminated and is waiting for its termination to accepted.
    3. return immediately with an error -1 if it does not have any child process.
- when successful, `wait()` returns the PID of the terminating child process.
- Some bit manipulation macros have been defined to deal with the value int the variable status (You need to include `<sys/wait.h>`)
    1. **WIFEXITED(status)**: return true for normal child.
    2. **WEXITSTATUS(status)**: used only when `WIFEXITED(status)` is true, it returns the exit status as an integer(0 - 255).
    3. **WIFSIGNALED(status)**: true for abnormal child termination.
    4. **WTERMSIG(status)**: used only when `WIFSIGNALED(status)` is true, it returns the signal number that caused the abnormal termination of the child process.

- ***NOTE***: Total status = 0 - 255 (To calculate the status more than 255 = Code % 256).

1. **waitpid()**
   - It allows parent to wait for a specific child to terminate and to accept it child’s termination code.

```
#include<sys/types.h>
#include<unistd.h>
pid_t waitpid(pid_t pid, int* status, int options);
```

- returns the PID of the specific child process (upon its termination)
- return immediately with an error -1 if doesn’t have any child process.
- ***NOTE***: `wait(&status)` is equivalent to `waitpid(-1, &status, 0)`.

1. **Orphan and Zombie Processes**
   - A process that terminates does not actually leave the system before the parent process accepts its return.
   - **Situations**
     1. Parent exits(e.g. the parent has been killed prematurely) while its children are still alive. The children becomes **orphans.**
     2. Parent is not in position to accept the exit and the termination code of the child process (parent is in an infinite loop). The children becomes **zombies.**
   - because some process must accept their return codes, the kernel simply changes their **PPID to 1** (init process) in the absense of a parent process.
   - Orphan processes are systematically adopted by the process init **(PID of init is 1)** and init accepts all its children returns.
   - When parent processes is not able to accept the termination code of their child processes, the

children becomes and remain in the system's process table waiting for the acceptance of their return. However, they loose their resources.
- too many zombie processes can require the intervention of the system administrator.
- ***NOTES***
  1. Zombie process are called `[defunct process]`.
  2. All children becomes a **zombie process** for a instance.
  3. Kill the parent to remove the zombie parent.

2. **exec()**
   - Family of system calls allows a process to **replace its current code, data and stack with those of another program (PID remains the same).**
   - The `exec()` functions return only if an error has occurred. The return value is -1.
   - **Functions**
     1. **execl()**
        - int execl(const char* path, [const char *argi,]+, NULL);
        - e.g.`int execl("/bin/ls", "/bin/ls", "-1", NULL);  // absolute path name is required.`
     2. **execlp()**
        - int execlp(const char* path, [const char * argi,]+, NULL);
        - e.g. `int execl("ls", "ls", "-1", NULL); // absolute pathname is not required.`
     3. **execv()**
        - int execv(const char* path, char* const argv[]);
        - e.g. `char *args[] = {"ls", "-1", NULL}; int execv("/bin/ls", args); //absolute pathname is required.`
     4. **execvp()**
        - int execvp(const char* path, char* const argv[]);
        - e.g. `char *args[] = {"ls", "-1", NULL}; int execvp("ls", args); //absolute pathname is not required.`

```
// ALL LECTURE 3 CODE
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
```

```
int main() {
  int i = fork();

  if (i == 0) {
    printf("This is Child\n");
    printf("i: %d\n", i);
    exit(0);
  } else if (i < 0) {
    printf("Failed!\n");
  } else {
    printf("This is Parent\n");
    printf("i: %d\n", i);
    sleep(10);
  }

  return 0; // This will create a <defunct> process.
}

#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main() {
  int cpid1, cpid2, gc1_pid, gc2_pid;
  int status;
  cpid1 = fork();

  if (cpid1 == 0) {
    gc1_pid = fork();
    if (gc1_pid == 0) {
      printf("GC1 will now differenciate\n");
      char *args[] = {"ls", "-1", "/Users/0xskaper/mimir/
AdvanceSystems", NULL}; // ls -1 /Users/0xskaper/mimir/
AdvanceSystems
      execvp("ls", args);
      exit(0);
    } else if (gc1_pid < 0) {
      perror("Fork Failed!");
      exit(1);
    } else {
      printf("C1 waiting fo GC1\n");
      waitpid(gc1_pid, &status, 0);
      if (WIFEXITED(status)) {
        printf("GC1 Exit Status: %d\n", WEXITSTATUS(status)); //
GC1 Exit Status: 0
      }
      if (WIFSIGNALED(status)) {
        printf(" GC1 Exit Signaled Status: %d\n",
WTERMSIG(status));
      }
      exit(0);
    }
  }

  cpid2 = fork();
  if (cpid2 == 0) {
    gc2_pid = fork();
    if (gc2_pid == 0) {
      printf("GC2 will now differenciate to pwd\n");
      raise(SIGSEGV); // This will cause a signal to be sent to the
parent.
      char *args[] = {"pwd", NULL};
      if (chdir("/Users/0xskaper/mimir") < 0) {
        perror("Chdir failed in GC2\n");
        exit(1);
      }
      execvp("pwd", args);
      exit(0);
    } else if (gc2_pid < 0) {
      perror("fork failed!");
      exit(1);
    } else {
      printf("C2 waiting for GC2\n");
      waitpid(gc2_pid, &status, 0);

      if (WIFEXITED(status)) {
        printf("GC2 Exit Status: %d\n", WEXITSTATUS(status));
      }
      if (WIFSIGNALED(status)) {
        printf("GC2 Exit Signaled Status: %d\n",
WTERMSIG(status)); // GC2 Exit Signaled Status: 11
      }
      exit(0);
    }
  }
  waitpid(cpid1, &status, 0);
  waitpid(cpid2, &status, 0);
  return 0;
}
```

Lecture 4

1. **Signals**
   - A signal is a software interrupt delivered to a process.
   - It used for Inter-process communication.
   - **Types of Signals**
     1. **Synchronous:** generated by the kernel in response to a particular event.
     2. **Asynchronous:** generated by the kernel in response to an event that occurs outside the process.
   - A process can also to send a signal to another process, as long as the sender has the permission to do so.
   - **Signal Number:** is a non-negative integer.
   - Every signal name is prefixed with **SIG.**
   - **Signal Handling**
     1. **Default:** the default action is taken by the kernel when a signal is received.
     2. **User-handler:** the user has defined a handler for the signal.
     3. **Ignore:** the signal is ignored. (SIGKILL and SIGSTOP cannot be ignored)
   - Default action for a signal can be:
     1. **Terminate:** the process is terminated.
     2. **Terminate and Core Dump:** the process is terminated and a core dump is generated.
     3. **Ignore:** the signal is ignore.
     4. **Suspension:** the process is suspended.
     5. **Resume:** the process is resumed.
   - **List**

#	Signal Name	Default Action	Comment	POSIX
1	SIGHUP	Terminate	Hang up controlling terminal or process	Yes
2	SIGINT	Terminate	Interrupt from keyboard, Control-C	Yes
3	SIGQUIT	Dump	Quit from keyboard, Control-\	Yes
4	SIGILL	Dump	Illegal instruction	Yes
5	SIGTRAP	Dump	Breakpoint for debugging	No
6	SIGABRT	Dump	Abnormal termination	Yes
6	SIGIOT	Dump	Equivalent to SIGABRT	No
7	SIGBUS	Dump	Bus error	No
8	SIGFPE	Dump	Floating-point exception	Yes
9	SIGKILL	Terminate	Forced-process termination	Yes
10	SIGUSR1	Terminate	Available to processes	Yes
11	SIGSEGV	Dump	Invalid memory reference	Yes
12	SIGUSR2	Terminate	Available to processes	Yes
13	SIGPIPE	Terminate	Write to pipe with no readers	Yes
14	SIGALRM	Terminate	Real-timer clock	Yes
15	SIGTERM	Terminate	Process termination	Yes
16	SIGSTKFLT	Terminate	Coprocessor stack error	No
17	SIGCHLD	Ignore	Child process stopped or terminated or got a signal if traced	Yes
18	SIGCONT	Continue	Resume execution, if stopped	Yes
19	SIGSTOP	Stop	Stop process execution, Ctrl-Z	Yes
20	SIGTSTP	Stop	Stop process issued from tty	Yes
21	SIGTTIN	Stop	Background process requires input	Yes
22	SIGTTOU	Stop	Background process requires output	Yes
23	SIGURG	Ignore	Urgent condition on socket	No
24	SIGXCPU	Dump	CPU time limit exceeded	No
25	SIGXFSZ	Dump	File size limit exceeded	No
26	SIGVTALRM	Terminate	Virtual timer clock	No
27	SIGPROF	Terminate	Profile timer clock	No
28	SIGWINCH	Ignore	Window resizing	No
29	SIGIO	Terminate	I/O now possible	No
29	SIGPOLL	Terminate	Equivalent to SIGIO	No
30	SIGPWR	Terminate	Power supply failure	No
31	SIGSYS	Dump	Bad system call	No
31	SIGUNUSED	Dump	Equivalent to SIGSYS	No

1. **alarm()**
    - It is used to set a timer that generates a signal when it expires.

```c
#include<unistd.h>

unsigned int alarm(unsigned int seconds);
```

- **seconds**: the number of seconds before the alarm signal is generated.
- **Return**: the number of seconds remaining before the alarm was scheduled to be delivered.
- **NOTE**: if the process has previously set an alarm, the previous alarm is cancelled.
- **NOTE**: n = 0 all pending alarm/s will be cancelled.

1. **signal()**
   - It is used to set a signal handler for a particular signal.

```
#include<signal.h>

void (*signal(int signo, void (*handler)(int)))(int);
```

- returns the previous signal handler or -1 on error.
- **signo**: the signal number.
- **handler**: the signal handler.
  1. **SIG_DFL**: default action. (macro not a signal)
  2. **SIG_IGN**: ignore the signal. (macro not a signal)
  3. **user-defined function**: the signal handler is a function that takes an integer as an argument and returns void.
- **NOTE**: the signal handler is a function that takes an integer as an argument and returns void.

1. **pause()**
   - It suspends the calling process until a signal is delivered.
   - **Function**:

```
#include<unistd.h>

int pause(void);
```

- returns -1 and sets `errno` to `EINTR` (interrupted system call) when a signal is delivered.
- **NOTE**: the process is suspended until a signal is delivered.
- Exception applied to SIGCHLD, SIGCONT, and SIGURG.

- if a signal terminates the process (in absence of a signal handler), it never returns.
- It only returns when a signal handler is defined for the signal.

```c
// ALL LECTURE 4 CODE
#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
#include <unistd.h>

int main(int argc, char *argv[]) {

  int a = alarm(4); // a = 0
  printf("a: %d\n", a);

  int b = alarm(10); // b = 4
  printf("b: %d\n", b);

  while (1) {
    printf("Sec\n");
    sleep(1);
  }
  printf("Exiting on Alarm\n");
  exit(0);
}

#include <stdio.h>
#include <stdlib.h>
#include <sys/signal.h>
#include <unistd.h>

// Return value and other features of the alarm() system call

int main(int argc, char *argv[]) {

  alarm(6);
  int ret = alarm(4); // ret = 6
  printf("alarm() is: %d ", ret);

  while (1) {
    printf("sec\n");
    sleep(1);
  }
  printf("Exiting on Alarm\n");
  exit(0);
}

#include <stdio.h>
#include <sys/signal.h>
#include <unistd.h>

void CtrHandler(int signum) { printf("%d is pressed\n", signum); }

int main(int argc, char *argv[]) {
  void (*oldHandler1)(); // to save default handlers for CTR-C and
CTR-Z
  void (*oldHandler2)(); // to save default handlers for CTR-C and
CTR-Z
  void (*newHandler1)(); // to save new handlers for CTR-C and CTR-
Z
  void (*newHandler2)(); // to save new handlers for CTR-C and CTR-
Z

  oldHandler1 = signal(SIGINT, CtrHandler);  // CTR-C
  oldHandler2 = signal(SIGTSTP, CtrHandler); // CTR-Z

  for (int i = 1; i <= 10; i++) {
    printf("I am not sensitive to CTR-C/CTR-Z\n");
    sleep(1);
  }

  newHandler1 = signal(SIGINT, oldHandler1);  // restore default
  newHandler2 = signal(SIGTSTP, oldHandler2); // restore default

  for (int i = 1; i <= 10; i++) {
    printf("I am sensitive to CTR-C/CTR-Z\n");
    sleep(1);
  }

  oldHandler1 = signal(SIGINT, newHandler1);  // ignore CTR-C
  oldHandler2 = signal(SIGTSTP, newHandler2); // ignore CTR-Z

  for (int i = 1; i <= 10; i++) {
    printf("I am not sensitive to CTR-C/CTR-Z\n");
    sleep(1);
  }
```

```
}

#include <stdio.h>
#include <sys/signal.h>
#include <unistd.h>

void Handler(int signo) { printf("\nIn the handler\n"); }

int main(int argc, char *argv[]) {

  signal(SIGALRM, Handler); // install the handler

  alarm(5);

  printf("Pausing\n");

  int i = pause();

  printf("Resuming\n");

  printf("The return value of pause(): %d\n",
         i); // if handler -1 else never returns
}

#include <signal.h>
#include <stdio.h>
#include <unistd.h>

void handler(int sig) { printf("Time\n"); }

int main() {
  int ret = alarm(10);
  printf("%d\n", ret);
  sleep(9);
  ret = alarm(3);
  printf("%d\n", ret);
  return 0;
}
```