CS 332/532 Systems Programming

Lecture 26

Review

Professor : Mahmut Unan – UAB CS

Announcement

- Exam 2
  - 11/01/2024 - Friday Lecture Time

- Cumulative Exam

- Canvas Quiz/lockdown browser
- Similar to Exam 1

Standard I/O Library

- This library is specified by ISO C standard because it has been implemented on many OS
- The standard I/O library handles such details as buffer allocation and performing I/O in optimal-sized chunks, obviating our need to worry about using the correct block size

Recall - File Descriptor

- When a file opened
  - nonnegative int assigned
  - this int is used in all operations

- Streams
  - with the standard I/O library, the discussion centers on streams
  - Open or Create a file → associate with a stream

I/O Stream

- *Open I/O Stream:* The standard I/O stream allows you to open a file in read, write, or append modes. This mode can be combined in a single open function call (see Figure 5.2 in Section 5.5 of the textbook for a complete list of options that can be specified). For example:

```
FILE *fptr;
fptr = fopen("listings.csv", "r+");
```

type	Description	open(2) Flags
`r` or `rb`	open for reading	`O_RDONLY`
`w` or `wb`	truncate to 0 length or create for writing	`O_WRONLY|O_CREAT|O_TRUNC`
`a` or `ab`	append; open for writing at end of file, or create for writing	`O_WRONLY|O_CREAT|O_APPEND`
`r+` or `r+b` or `rb+`	open for reading and writing	`O_RDWR`
`w+` or `w+b` or `wb+`	truncate to 0 length or create for reading and writing	`O_RDWR|O_CREAT|O_TRUNC`
`a+` or `a+b` or `ab+`	open or create for reading and writing at end of file	`O_RDWR|O_CREAT|O_APPEND`

Figure 5.2 The *type* argument for opening a standard I/O stream

Input / Output Stream:

- ***Input Stream*:** The standard I/O stream allows us to read from the open file. These functions allow us to read a file character by character – getchar(), line by line – fgets(), or with specific size – fread()
- ***Output Stream*:** The standard I/O stream allow you to write to an open file. These functions allow you to write to a file character by character – putchar(), line by line – fputs(), or with specific size – fwrite()

Exercise 1

- Now let’s use these functions and write a program. We will use APIs available in Linux and C to develop different versions of this program

getc()

```
int getc(FILE *stream)
```

- Gets the next character (an unsigned char) from the specified stream
- Advances the position indicator for the stream.

getline1.c

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

int getLine(FILE *fp, char *line);

int main(int argc, char** argv) {
    char *str;
    FILE *fp;
    int n;

    str = malloc(sizeof(char)*BUFSIZ);
    fp = fopen( filename: argv[1],  mode: "r");
    if (fp == NULL) {
        printf("Error opening file %s\n", argv[1]);
        exit(-1);
    }
    while ( (n = getLine(fp, str)) > 0)
        printf("%d: %s\n", n, str);
    fclose(fp);
    return 0;
}

int getLine(FILE *fp, char *line) {
    int c, i=0;
    while ((c = getc(fp)) != '\n' && c != EOF)
        line[i++] = c;
    line[i] = '\0';
    return i;
}
```

```
Some text line 1
Line 2
l3
```

```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline1 getline1.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline1 test.txt
16: Some text line 1
6: Line 2
2: l3
```

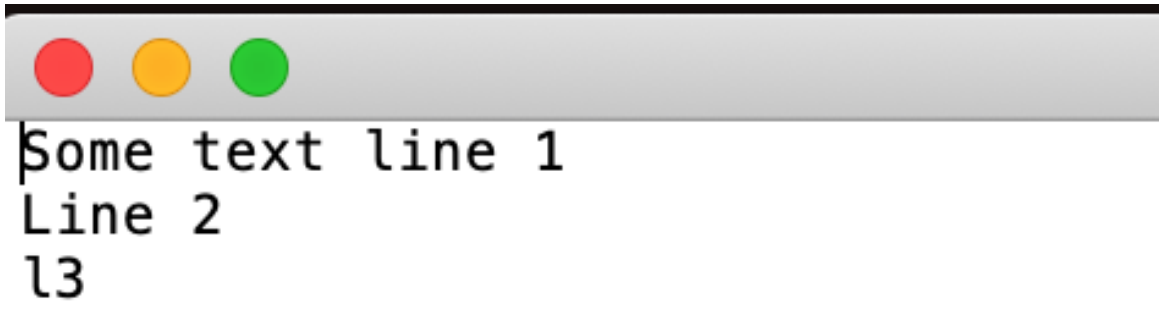
getline()

```
ssize_t getline(char **lineptr, size_t *n, FILE *stream);
```

- **getline**() reads an entire line from *stream*, storing the address of the buffer containing the text into **lineptr*. The buffer is null-terminated and includes the newline character, if one was found.
- On success, **getline**() returns the number of characters read, including the delimiter character, but not including the terminating null byte ('\0')

getline2.c

```c

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

int main(int argc, char** argv) {
    char *line=NULL;
    FILE *fp;
    size_t maxlen=0;
    ssize_t n;

    printf("BUFSIZ = %d\n", BUFSIZ);

    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
        printf("Error opening file %s\n", argv[1]);
        exit(-1);
    }
    while ( (n = getline(&line, &maxlen, fp)) > 0)
        printf("%ld[%ld]: %s\n", n, maxlen, line);

    fclose(fp);
    return 0;
}
```




```
(base) mahmutunan@MacBook-Pro lecture15 % ./getline2 test.txt
BUFSIZ = 1024
17[32]: Some text line 1

7[32]: Line 2

2[32]: l3
```

getdelim()

- `ssize_t getdelim(char **lineptr, size_t *n, int delim, FILE *stream);`

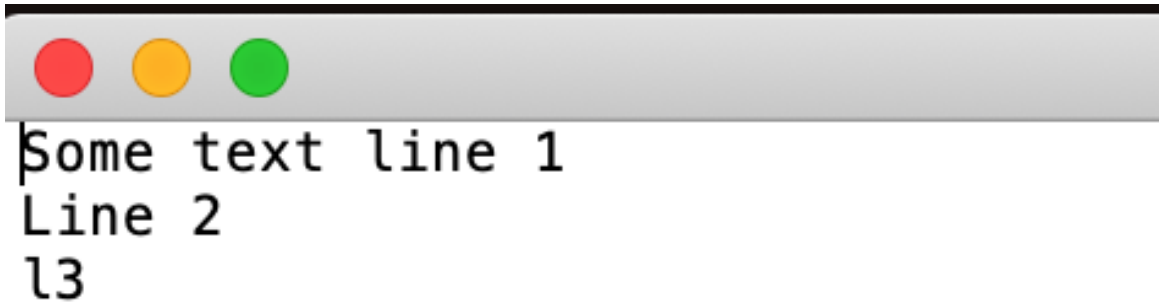
- getdelim() works like getline(), except that a line delimiter other than newline can be specified as the *delimiter* argument.

- getdelim() returns the number of characters read, including the delimiter character, but not including the terminating null byte

https://linux.die.net/man/3/getdelim

getline3.c

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

int main(int argc, char** argv) {
    char *line=NULL;
    FILE *fp;
    size_t maxlen=0;
    ssize_t n;

    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
        printf("Error opening file %s\n", argv[1]);
        exit(-1);
    }
    while ( (n = getdelim(&line, &maxlen, delimiter: ' ', fp)) > 0)
        printf("%ld: %s\n", n, line);

    fclose(fp);
    return 0;
}

```




```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline3 getline3.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline3 test.txt
5: Some
5: text
5: line
7: 1
Line
4: 2
l3
(base) mahmutunan@MacBook-Pro lecture15 % _
```

gets()

- `char *gets(char *str)`

- The C library function char *gets(char *str) reads a line from stdin and stores it into the string pointed to by str.
- It stops when either the newline character is read or when the end-of-file is reached, whichever comes first.

- It reads string from standard input and prints the entered string, but it suffers from Buffer Overflow as gets() doesn't do any array bound testing.
- gets() keeps on reading until it sees a newline character.
- To avoid Buffer Overflow, fgets() should be used instead of gets() as fgets() makes sure that not more than MAX_LIMIT characters are read.

https://www.geeksforgeeks.org/gets-is-risky-to-use/

fgets()

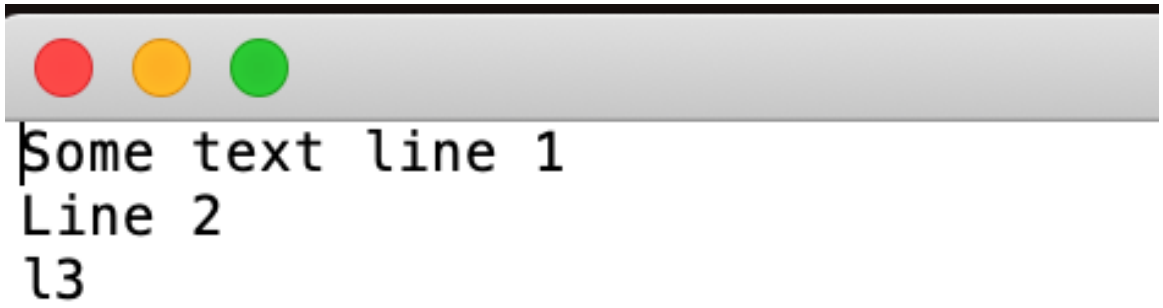
- `char *fgets(char *str, int n, FILE *stream)`

- **str** – This is the pointer to an array of chars where the string read is stored.
- **n** – This is the maximum number of characters to be read (including the final null-character). Usually, the length of the array passed as str is used.
- **stream** – This is the pointer to a FILE object that identifies the stream where characters are read from.
- On success, the function returns the same str parameter. If the End-of-File is encountered and no characters have been read, the contents of str remain unchanged and a null pointer is returned

https://www.tutorialspoint.com/c_standard_library/c_function_fgets.htm#:~:text=The%20C%20library%20function%20char,is%20reached%2C%20whichever%20comes%20first.

getline4.c

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

int main(int argc, char** argv) {
    char *line;
    FILE *fp;

    line = malloc(sizeof(char)*BUFSIZ);

    if ((fp = fopen( filename: argv[1], mode: "r")) == NULL) {
        fprintf(stderr,"Error opening file %s\n", argv[1]);
        exit(-1);
    }
    while ( fgets(line, BUFSIZ, fp) != NULL )
        fprintf(stdout,"%ld: %s\n", strlen(line), line);

    fclose(fp);
    return 0;
}
```




```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline4 getline4.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline4 test.txt
17: Some text line 1

7: Line 2

2: l3
(base) mahmutunan@MacBook-Pro lecture15 %
```

fscanf()

- `int fscanf(FILE *stream, const char *format, ...)`

Conversion type	Description
`d,i`	signed decimal
`o`	unsigned octal
`u`	unsigned decimal
`x,X`	unsigned hexadecimal
`f,F`	`double` floating-point number
`e,E`	`double` floating-point number in exponential format
`g,G`	interpreted as f, F, e, or E, depending on value converted
`a,A`	`double` floating-point number in hexadecimal exponential format
`c`	character (with `l` length modifier, wide character)
`s`	string (with `l` length modifier, wide character string)
`p`	pointer to a `void`
`n`	pointer to a signed integer into which is written the number of characters written so far
`%`	a % character
`C`	wide character (XSI option, equivalent to `lc`)
`S`	wide character string (XSI option, equivalent to `ls`)

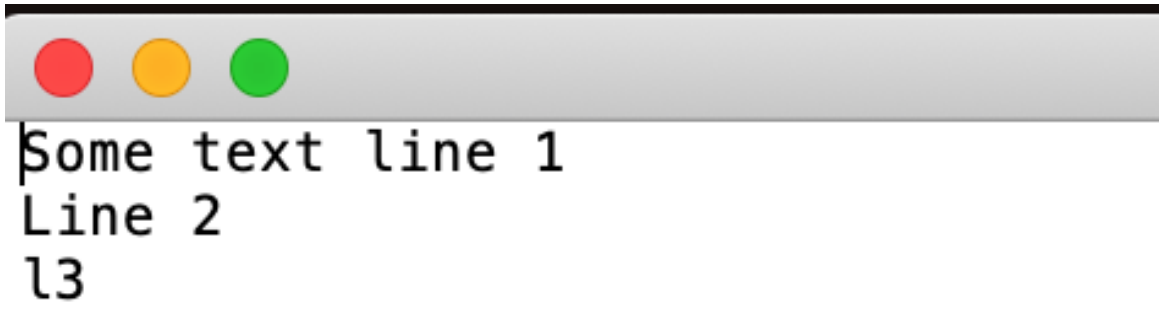
Figure 5.9 The conversion type component of a conversion specification

Conversion type	Description
`d`	signed decimal, base 10
`i`	signed decimal, base determined by format of input
`o`	unsigned octal (input optionally signed)
`u`	unsigned decimal, base 10 (input optionally signed)
`x,X`	unsigned hexadecimal (input optionally signed)
`a,A,e,E,f,F,g,G`	floating-point number
`c`	character (with `l` length modifier, wide character)
`s`	string (with `l` length modifier, wide character string)
`[`	matches a sequence of listed characters, ending with `]`
`[^`	matches all characters except the ones listed, ending with `]`
`p`	pointer to a `void`
`n`	pointer to a signed integer into which is written the number of characters read so far
`%`	a % character
`C`	wide character (XSI option, equivalent to `lc`)
`S`	wide character string (XSI option, equivalent to `ls`)

Figure 5.10 The conversion type component of a conversion specification

getline5.c

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

int main(int argc, char** argv) {
    char *line;
    FILE *fp;

    line = malloc(sizeof(char)*BUFSIZ);

    if ((fp = fopen( filename: argv[1],  mode: "r")) == NULL) {
        printf("Error opening file %s\n", argv[1]);
        exit(-1);
    }
    while ( fscanf(fp, "%s", line) != EOF )
        printf("%ld: %s\n", strlen(line), line);

    fclose(fp);
    return 0;
}
```




```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline5 getline5.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline5 test.txt
4: Some
4: text
4: line
1: 1
4: Line
1: 2
2: l3
```

fprintf()

```
int fprintf(FILE *stream, const char *format, ...)
```

stream

The stream where the output will be written.

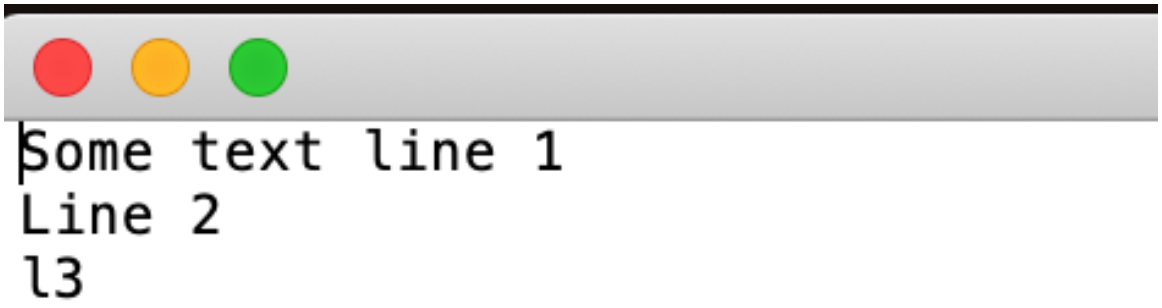
format

Describes the output as well as provides a placeholder to insert the formatted string. Here are a few examples:

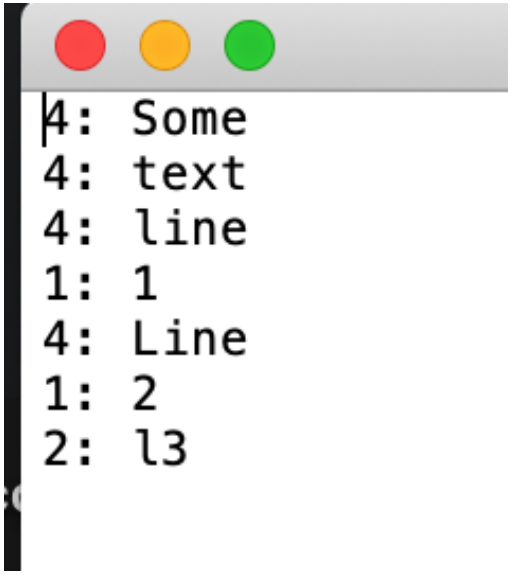
Format	Explanation	Example
%d	Display an integer	10
%f	Displays a floating-point number in fixed decimal format	10.500000
%.1f	Displays a floating-point number with 1 digit after the decimal	10.5
%e	Display a floating-point number in exponential (scientific notation)	1.050000e+01
%g	Display a floating-point number in either fixed decimal or exponential format depending on the size of the number (will not display trailing zeros)	10.5

getline6.c

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

int main(int argc, char** argv) {
    char *line;
    FILE *fp, *fpout;

    line = malloc(sizeof(char)*BUFSIZ);

    if ((fp = fopen( filename: argv[1],  mode: "r")) == NULL) {
        fprintf(stderr,"Error opening file %s\n", argv[1]);
        exit(-1);
    }
    if ((fpout = fopen( filename: argv[2],  mode: "w")) == NULL) {
        fprintf(stderr,"Error opening file %s\n", argv[2]);
        exit(-1);
    }
    while ( fscanf(fp, "%s", line) != EOF )
        fprintf(fpout,"%ld: %s\n", strlen(line), line);

    fclose(fp);
    fclose(fpout);
    return 0;
}
```




```
(base) mahmutunan@MacBook-Pro lecture15 % gcc -o getline6 getline6.c
(base) mahmutunan@MacBook-Pro lecture15 % ./getline6 test.txt output.txt
(base) mahmutunan@MacBook-Pro lecture15 % _
```



Example 2

- We will now write a program to read a comma separated file (“listing.csv”) and use the C structures to store and display the data on the console.
- The sample input file used is :

	A	B	C	D	E	F	G	H	I	J	K	L	M
1	id	host_id	host_name	neighbourho	neighbourho	latitude	longitude	room_type	price	minimum_ni	number_of_	calculated_h	availability_365
2	2015	2217	Ian	Mitte	Brunnenstr. S	52.5345373	13.4025569	Entire home/	60	4	118	4	141
3	2695	2986	Michael	Pankow	Prenzlauer B	52.5485128	13.4045528	Private room	17	2	6	1	0
4	3176	3718	Britta	Pankow	Prenzlauer B	52.5349962	13.4175787	Entire home/	90	62	143	1	220
5	3309	4108	Jana	Tempelhof -	SchVðneberg	52.4988549	13.3490645	Private room	26	5	25	1	297
6	7071	17391	Bright	Pankow	Helmholtzpla	52.5431573	13.4150911	Private room	42	2	197	1	26
7	9991	33852	Philipp	Pankow	Prenzlauer B	52.5330308	13.4160468	Entire home/	180	6	6	1	137
8	14325	55531	Chris + Olive	Pankow	Prenzlauer B	52.5478464	13.4055622	Entire home/	70	90	23	3	129
9	16401	59666	Melanie	Friedrichshai	Frankfurter A	52.510514	13.4578502	Private room	120	30	0	1	365
10	16644	64696	Rene	Friedrichshai	nVðrdliche Lu	52.5047923	13.4351019	Entire home/	90	60	48	2	159
11	17409	67590	Wolfram	Pankow	Prenzlauer B	52.5290709	13.4128434	Private room	45	3	279	1	42
12	17904	68997	Matthias	NeukVðlln	ReuterstraVü	52.4954763	13.4218213	Entire home/	49	5	223	1	232
13	20858	71331	Marc	Pankow	Prenzlauer B	52.5369524	13.407615	Entire home/	129	3	56	1	166
14	21869	64696	Rene	Friedrichshai	nVðrdliche Lu	52.5027333	13.4346199	Entire home/	70	60	60	2	129
15	22415	86068	Kiki	Friedrichshai	sVºdliche Lui	52.4948506	13.4285006	Entire home/	98	3	61	2	257
16	22677	87357	Ramfis	Mitte	Brunnenstr. S	52.5343484	13.4055765	Entire home/	160	3	223	1	228
17	23834	94918	Tanja	Friedrichshai	Tempelhofer	52.4897144	13.3797476	Entire home/	65	60	96	1	275
18	24569	99662	Dominik	Pankow	Prenzlauer B	52.5307909	13.4180844	Entire home/	90	3	18	2	3
19	25653	99662	Dominik	Pankow	Prenzlauer B	52.5302587	13.419467	Entire home/	90	4	5	2	15
20	26543	112675	Terri	Pankow	Helmholtzpla	52.5440624	13.4213765	Entire home/	197	3	163	1	336
21	28156	55531	Chris + Olive	Pankow	Prenzlauer B	52.5467194	13.405117	Entire home/	70	90	28	3	191
22	28268	121580	Elena	Friedrichshai	Frankfurter A	52.5133852	13.4699475	Entire home/	90	5	30	1	55
23	28711	84157	Emanuela	NeukVðlln	ReuterstraVü	52.4861061	13.434817	Entire home/	60	2	1	10	341
24	29279	54283	Marine	Pankow	Helmholtzpla	52.5417876	13.4238832	Entire home/	130	60	69	3	221

listing.c

- The given file has 13 different attributes, and these attributes can be divided into three different datatypes: integer, character array, and float.
- And collectively we can create a C structure to represent these attributes and then create an array of such structures to store multiple entities.
- 1. Define a structure called listing with all attributes as individual members of the struct listing.

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

#define LINESIZE 1024

struct listing {
    int id, host_id, minimum_nights, number_of_reviews, calculated_host_listings_count,availability_365;
    char *host_name, *neighbourhood_group, *neighbourhood, *room_type;
    float latitude, longitude, price;
};
```

strtok

- `char *strtok(char *str, const char *delim);`
- `char *strtok_r(char *str, const char *delim, char **saveptr);`

- strtok, strtok_r - extract tokens from strings

- The strtok() function breaks a string into a sequence of zero or more nonempty tokens. On the first call to strtok(), the string to be parsed should be specified in *str*. In each subsequent call that should parse the same string, *str* must be NULL.

https://man7.org/linux/man-pages/man3/strtok.3.html

atoi

- atoi, atol, atoll - convert a string to an integer

- `int atoi(const char *nptr);`
- `long atol(const char *nptr);`
- `long long atoll(const char *nptr);`

- Return value: The converted value or 0 on error.

https://man7.org/linux/man-pages/man3/atoi.3.html

atof

- atof - convert a string to a double

```
double atof(const char *nptr);
```

RETURN VALUE : The converted value

https://man7.org/linux/man-pages/man3/atof.3.html

strdup

- strdup- duplicate a string

```
char *strdup(const char *s);
```

On success, the strdup() function returns a pointer to the duplicated string. It returns NULL if insufficient memory was available, with errno set to indicate the cause of the error.

https://www.man7.org/linux/man-pages/man3/strndup.3.html#:~:text=DESCRIPTION%20top,copies%20at%20most%20n%20bytes.

listing.c

- 2. Define a function which can help to parse each line in the file and return the above defined structure.
- For this task you need to learn the string tokenizer function (*strtok*) which is available in <*string.h*> header file.
- You can find out more about the *strtok* function by typing *man strtok*.
- You will notice that when you invoke the *strtok* function for the first time you provide the pointer to the character array and on subsequent invocations of *strtok* we use *NULL* as the argument.

listing.c

```c
struct listing getfields(char* line){
    struct listing item;

    item.id = atoi(strtok(line, ","));
    item.host_id = atoi(strtok(NULL, ","));
    item.host_name = strdup(strtok(NULL, ","));
    item.neighbourhood_group = strdup(strtok(NULL, ","));
    item.neighbourhood = strdup(strtok(NULL, ","));
    item.latitude = atof(strtok(NULL, ","));
    item.longitude = atof(strtok(NULL, ","));
    item.room_type = strdup(strtok(NULL, ","));
    item.price = atof(strtok(NULL, ","));
    item.minimum_nights = atoi(strtok(NULL, ","));
    item.number_of_reviews = atoi(strtok(NULL, ","));
    item.calculated_host_listings_count = atoi(strtok(NULL, ","));
    item.availability_365 = atoi(strtok(NULL, ","));

    return item;
}
```

listing.c

Now, create a function to display the items in the struct

```
36  void displayStruct(struct listing item) {
37      printf("ID : %d\n", item.id);
38      printf("Host ID : %d\n", item.host_id);
39      printf("Host Name : %s\n", item.host_name);
40      printf("Neighbourhood Group : %s\n", item.neighbourhood_group);
41      printf("Neighbourhood : %s\n", item.neighbourhood);
42      printf("Latitude : %f\n", item.latitude);
43      printf("Longitude : %f\n", item.longitude);
44      printf("Room Type : %s\n", item.room_type);
45      printf("Price : %f\n", item.price);
46      printf("Minimum Nights : %d\n", item.minimum_nights);
47      printf("Number of Reviews : %d\n", item.number_of_reviews);
48      printf("Calculated Host Listings Count : %d\n", item.calculated_host_listings_count);
49      printf("Availability_365 : %d\n\n", item.availability_365);
50  }
```

listing.c

- 3. Now use *fopen* function to open file in read only mode. Notice that the *fopen* function returns a pointer of *FILE* type (file pointer) unlike the *open* function that returns an *integer* value as the file descriptor.

```
52  int main(int argc, char* args[]) {
53      struct listing list_items[22555];
54      char line[LINESIZE];
55      int i, count;
56
57      FILE *fptr = fopen("listings.csv", "r");
58      if(fptr == NULL){
59          printf("Error reading input file listings.csv\n");
60          exit (-1);
61      }
```

listing.c

- 4. Then loop through till the end of file (use fgets function) and store all data in the array of structures
- We can close the file using fclose function

```
62
63      count = 0;
64      while (fgets(line, LINESIZE, fptr) != NULL){
65          list_items[count++] = getfields(line);
66      }
67      fclose(fptr);
```

listing.c

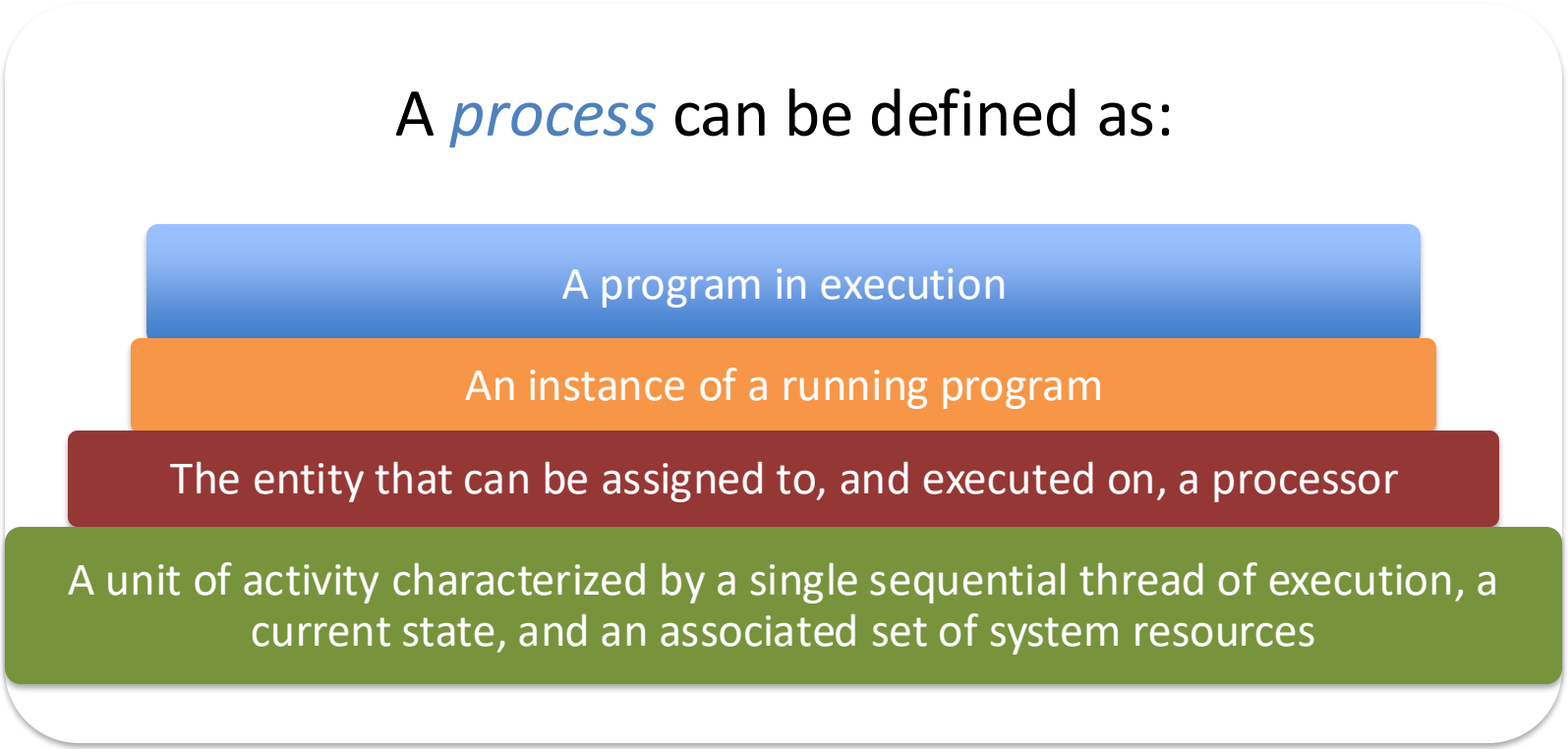
- 5. Now invoke the function to display the structure in a loop.

```
69      for (i=0; i<count; i++)
70          displayStruct(list_items[i]);
71
72      return 0;
73  }
74
```

Process Management

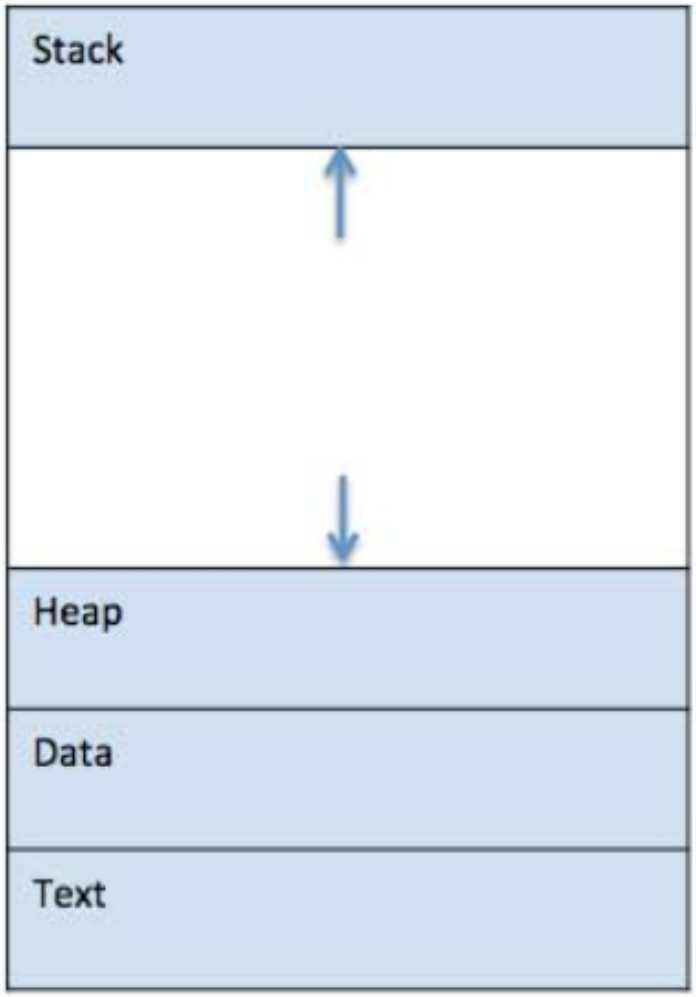
Process

- Fundamental to the structure of operating systems



Process

- When a program is loaded into the memory and it becomes a process, it can be divided into four sections
  - stack
  - heap
  - text
  - data.

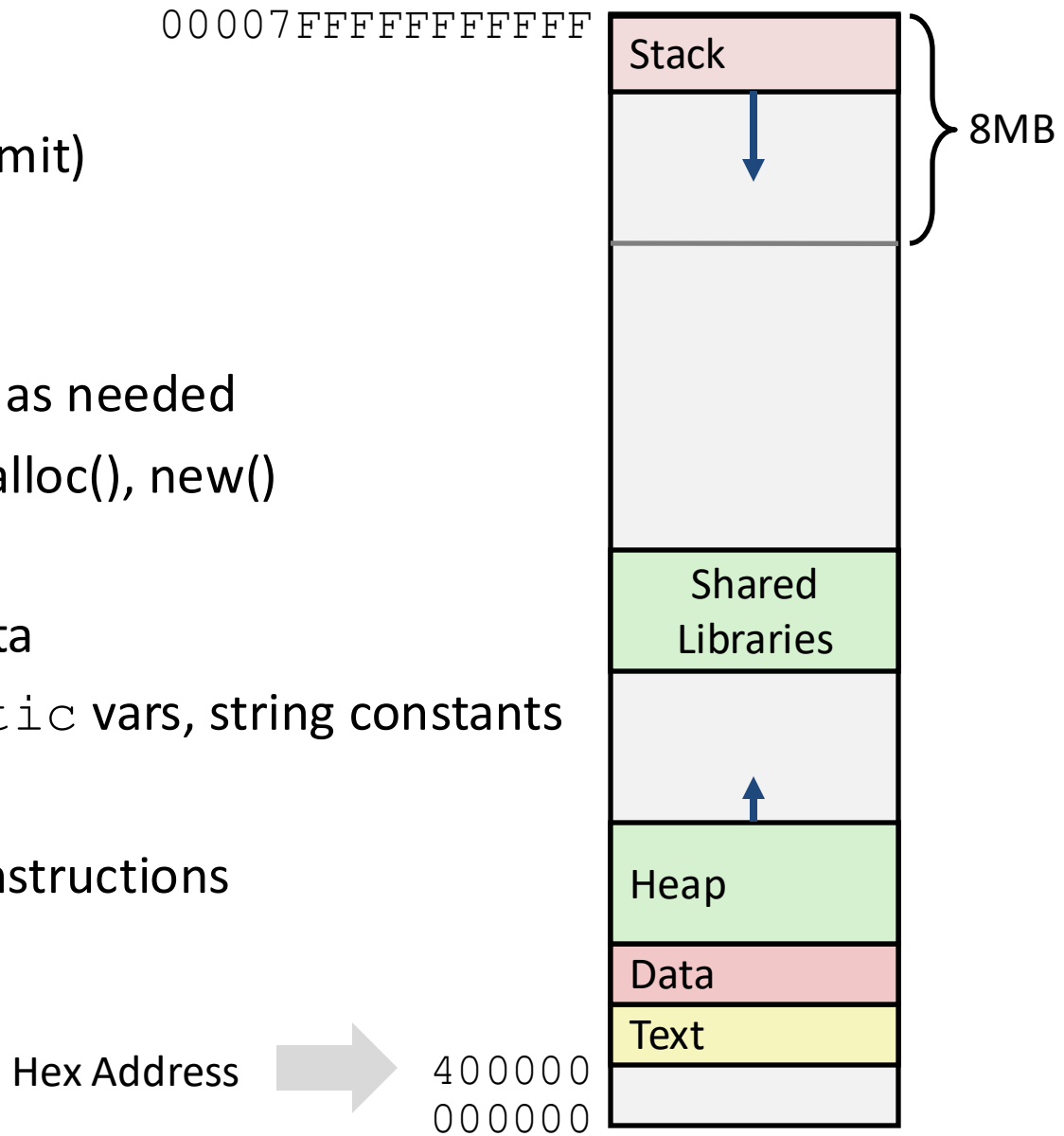


https://www.tutorialspoint.com/operating_system/os_processes.htm

x86-64 Linux Memory Layout

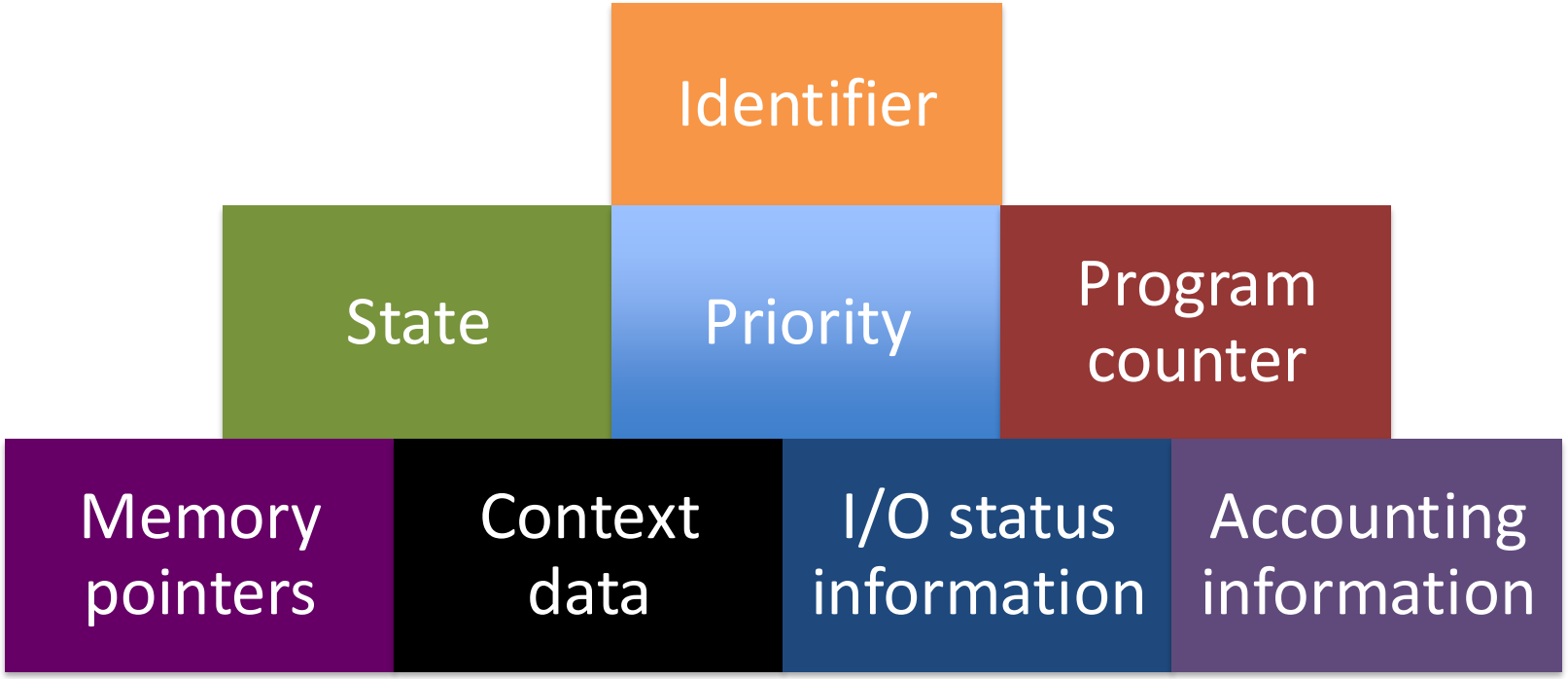
not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only



Process Elements

- While the program is executing, this process can be uniquely characterized by a number of elements, including:



Process Execution

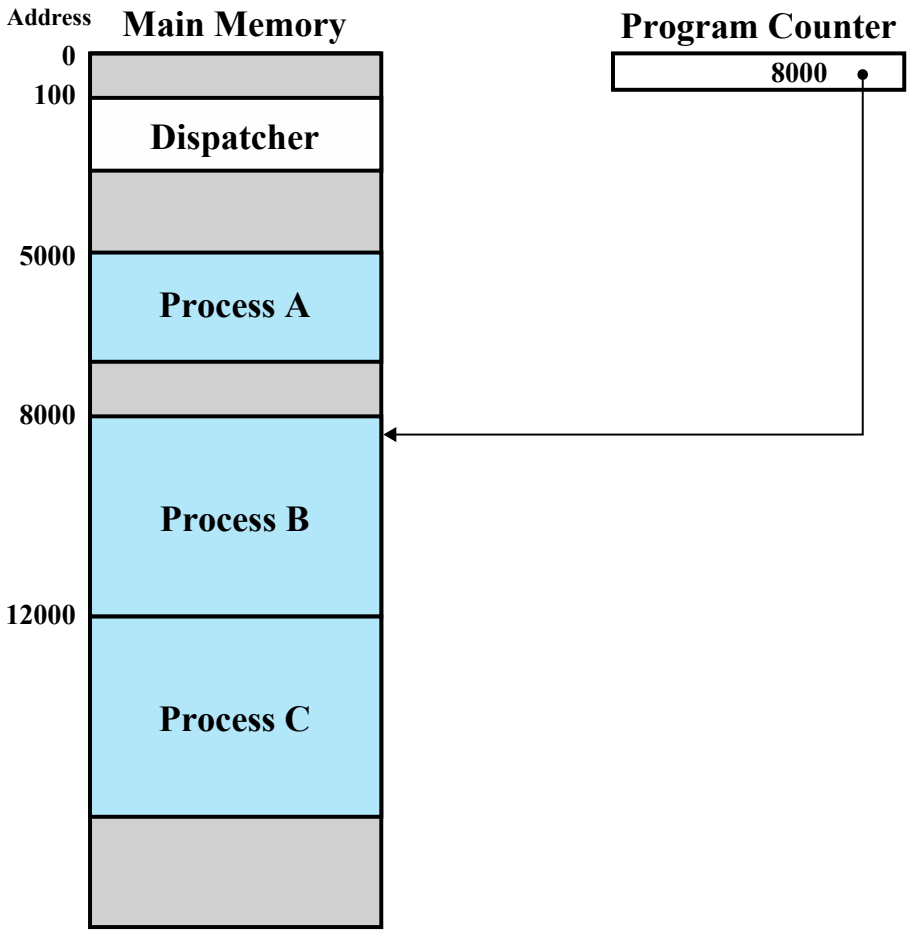


Figure 3.2 Snapshot of Example Execution (Figure 3.4) at Instruction Cycle 13

5000	8000	12000
5001	8001	12001
5002	8002	12002
5003	8003	12003
5004		12004
5005		12005
5006		12006
5007		12007
5008		12008
5009		12009
5010		12010
5011		12011

(a) Trace of Process A **(b) Trace of Process B** **(c) Trace of Process C**

5000 = Starting address of program of Process A
8000 = Starting address of program of Process B
12000 = Starting address of program of Process C

Figure 3.3 Traces of Processes of Figure 3.2

1	5000	27	12004
2	5001	28	12005
3	5002	----------	Timeout
4	5003	29	100
5	5004	30	101
6	5005	31	102
----------	Timeout	32	103
7	100	33	104
8	101	34	105
9	102	35	5006
10	103	36	5007
11	104	37	5008
12	105	38	5009
13	8000	39	5010
14	8001	40	5011
15	8002	----------	Timeout
16	8003	41	100
----------	I/O Request	42	101
17	100	43	102
18	101	44	103
19	102	45	104
20	103	46	105
21	104	47	12006
22	105	48	12007
23	12000	49	12008
24	12001	50	12009
25	12002	51	12010
26	12003	52	12011
		----------	Timeout

100 = Starting address of dispatcher program

Shaded areas indicate execution of dispatcher process;
first and third columns count instruction cycles;
second and fourth columns show address of instruction being executed

Figure 3.4 Combined Trace of Processes of Figure 3.2

Process Creation

Process spawning

- When the OS creates a process at the explicit request of another process

Parent process

- Is the original, creating, process

Child process

- Is the new process

Process Termination

- There must be a means for a process to indicate its completion
- A batch job should include a HALT instruction or an explicit OS service call for termination
- For an interactive application, the action of the user will indicate when the process is completed (e.g. log off, quitting an application)

Modes of Execution

User Mode

- Less-privileged mode
- User programs typically execute in this mode

System Mode

- More-privileged mode
- Also referred to as control mode or kernel mode
- Kernel of the operating system

Unix Processes

- The OS tracks processes through a five-digit ID number
  - **pid** or **process ID**.
- Each process in the system has a unique **pid**.
- If you want to list the running processes, use the **ps** (process status) command
  - to display the full option

    ```
    ps -f
    ```

    UID, PID, PPID, C, STIME, TTY, CMD, TIME…..

Unix Processes

- to stop a process
  - `kill`
  - `kill PID`
  - `kill -9 PID`
- Init Process
  - PID 1 and PPID 0
- Foreground Process
- Background Process

fork()

- fork - create a child process

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.

https://man7.org/linux/man-pages/man2/fork.2.html

fork()

- We can use the fork() system call to create a new process (referred to as the child process) which is an identical image of the calling process (referred to as the parent process).
- Here is the C interface for the fork() system call:

```
#include <unistd.h>

pid_t fork(void);
```

fork()

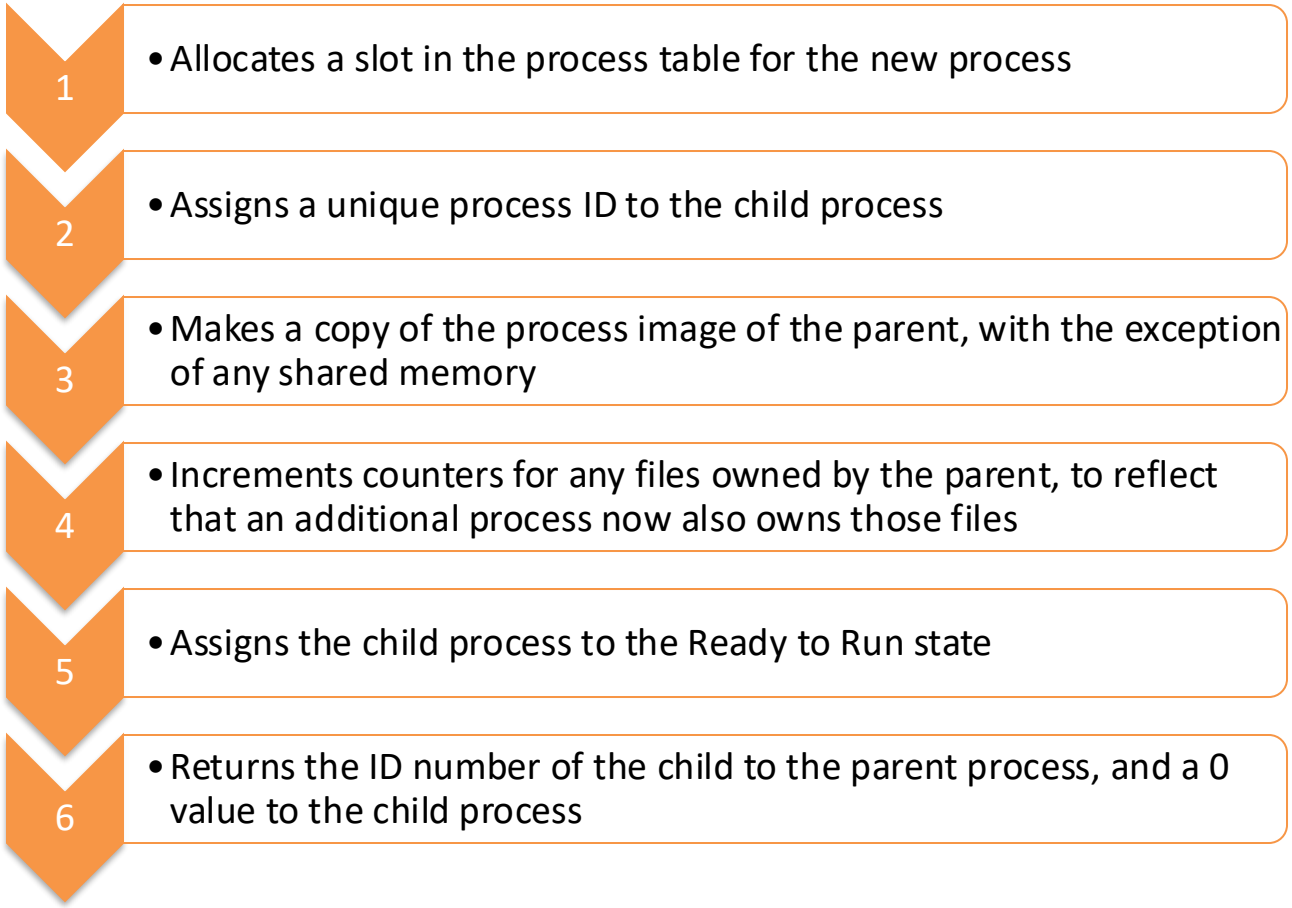
- If the fork() call is successful, then it returns the process ID of the child process to the parent process and returns 0 in the child process.
- fork() returns a negative value in the parent process and sets the corresponding errno variable (external variable defined in *errno.h*) if there is any error in process creation and the child process is not created.
- We can use perror() function (defined in *stdio.h*) to print the corresponding system error message. Look at the man page for perror to find out more about the perror() function.

fork()

- Once the parent process creates the child process, the parent process continues with its normal execution.
- If the parent process exits before the child process completes its execution and terminates, the child process will become a zombie process (*i.e.*, a process without a parent process).
- Alternatively, the parent process could wait for the child process to terminate using the wait() function.
- The wait() system call will suspend the execution of the calling process until one of the child process terminates and if there are no child processes available the wait() function returns immediately.

Process Control

- Process creation is by means of the kernel system call, *fork()*
- When a process issues a fork request, the OS performs the following functions:



Hello World!

```c
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

Hello World!

```c
int main() {
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
```

```c
int main() {
    fork();
    printf("Hello, World!\n");
    return 0;
}
```

```
Hello, World!
Hello, World!
```

Hello World!

```c
int main() {
    int processId = fork();
    printf("Hello, World! from= %d\n",processId);
    return 0;
}
```

```
Hello, World! from= 96037
Hello, World! from= 0
```

```c
int main() {
    printf("This is before the fork statement\n");
    fork();
    printf("After the FIRST fork\n");
    fork();
    printf("After the SECOND fork \n");
    fork();
    printf("After the THIRD fork \n");
    return 0;
}
```

```
After the FIRST fork
After the SECOND fork
After the FIRST fork
After the THIRD fork
After the SECOND fork
After the SECOND fork
After the SECOND fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
After the THIRD fork
```

```c
int main() {

    fork();
    fork();
    fork();
    fork();
    printf("4 forks will work 16 times\n");
    return 0;
}
```

```
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
4 forks will work 16 times
```

getpid()

```c
int main() {

    printf("before calling the fork %d\n",getpid());
    fork();
    printf("after calling the  FIRST fork %d\n",getpid());
    fork();
    printf("after calling the  SECOND fork %d\n",getpid());
}
```

```
before calling the fork 96717
after calling the  FIRST fork 96717
after calling the  SECOND fork 96717
after calling the  FIRST fork 96718
after calling the  SECOND fork 96719
after calling the  SECOND fork 96718
after calling the  SECOND fork 96720
```

wait()

wait, waitpid, waitid - wait for process to change state

```
 pid_t wait(int *wstatus);
 pid_t waitpid(pid_t pid, int *wstatus, int
options);
 int waitid(idtype_t idtype, id_t id, siginfo_t
*infop, int options);
```

https://www.man7.org/linux/man-pages/man2/waitid.2.html

wait()

- The wait() call returns the PID of the child process that terminated when successful, otherwise, it returns -1.
- The wait() call also sets an integer value that is passed as an argument to the function which can be inspected with various macros provided in <sys/wait.h> to determine how the child process completed (e.g., terminated normally, terminated by a signal).

wait() waitpid()

- If the calling process created more than one child process, we can use the waitpid() system call to wait on a specific child process to change state.
- A state change could be any one of the following events: the child was terminated; the child was stopped by a signal; or the child was resumed by a signal. Similar to wait(), waitpid() returns the PID of the child process that changed state when successful, otherwise, it returns -1.

wait() waitpid()

- Here are the C APIs for
  the wait() and waitpid() system calls:

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int
options);
```

```c
int main() {
    int processId = fork();
    int count;
    fflush(stdout);

    if (processId==0){
        count=1;
    }else{
        count=6;
    }
    if (processId!=0){
        wait();
    }
   int i;
    for (i=count;i<count+5;i++){
        printf("%d",i);
        fflush( stdout );
    }

}
```

```
12345678910
Process finished with exit code 0
```

https://www.youtube.com/watch?v=tcYo6hipaSA&ab_channel=CodeVault

Example 1

- We will create a sample program to illustrate how to use fork() to create a child process, wait for the child process to terminate, and display the parent and child process ID in both processes.

fork.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    pid = fork();
    if (pid == 0) { /* this is child process */
        printf("This is the child process, my PID is %ld and my parent PID is %ld\n",
            (long)getpid(), (long)getppid());
    } else if (pid > 0) { /* this is the parent process */
        printf("This is the parent process, my PID is %ld and the child PID is %ld\n",
            (long)getpid(), (long)pid);

        printf("Wait for the child process to terminate\n");
```

fork.c

```
18
19          printf("Wait for the child process to terminate\n");
20          wait(&status); /* wait for the child process to terminate */
21          if (WIFEXITED(status)) { /* child process terminated normally */
22              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23          } else { /* child process did not terminate normally */
24              printf("ERROR: Child process did not terminate normally!\n");
25              /* look at the man page for wait (man 2 wait) to
26                              determine how the child process was terminated */
27          }
28      } else { /* we have an error in process creation */
29          perror("fork");
30          exit(EXIT_FAILURE);
31      }
32
33      printf("[%ld]: Exiting program .....\n", (long)getpid());
34
35      return 0;
36  }
37
```

fork.c

```
(base) mahmutunan@MacBook-Pro lecture17 % ./exercise1
This is the parent process, my PID is 90695 and the child PID is 90696
Wait for the child process to terminate
This is the child process, my PID is 90696 and my parent PID is 90695
[90696]: Exiting program .....
Child process exited with status = 0
[90695]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture17 %
```

exec()

- execl, execlp, execle, execv, execvp, execvpe - execute a file
- The **exec**() family of functions replaces the current process image with a new process image.

https://man7.org/linux/man-pages/man3/exec.3.html

exec()

- Note that the child process is a copy of the parent process and control is split at the invocation of the fork() call between the parent and the child process.
- If we like the child process to execute a different program other than making a copy of the parent process, we can use the exec family of system calls to replace the current process image with a new one.

- Here is the C APIs for the exec family of system calls:

#include <unistd.h>

```
int execl(const char *pathname, const char
*arg, ...);
int execlp(const char *filename, const char
*arg, ...);
int execle(const char *pathname, const char
*arg, ..., char * const envp[]);
int execv(const char *pathname, char *const
argv[]);
int execvp(const char *filename, char *const
argv[]);
int execvpe(const char *filename, char *const
argv[], char *const envp[]);
```

- We will use the execl() to replace the child process created by fork().
- The execl() function takes as arguments the full pathname of the executable along with a pointer to an array of characters for each argument.
- Since we can have a variable number of arguments, the last argument is a null pointer.

p1.c

```c
#include <stdio.h>
#include <unistd.h>
int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    char *args[]={"Hello","CS","332",NULL};
    execv( path: "./p2",args);
    printf("we are not supposed to see this text");
    return 0;
}
```

p2

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n",argv[0],argv[1],argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30983
Hello from p2, process id= () 30983
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 % _
```

p1.c

- Let's modify it a little bit and fork the process

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

int main(int argc, char **argv) {
    printf("Hello from p1, process id= () %d\n", getpid());
    int pid=fork();
    int status;
    if (pid == 0) {
    char *args[] = {"Hello", "CS", "332", NULL};
    execv( path: "./p2", args);
    printf("we are not supposed to see this text");
   }
    else if(pid>0){
        wait(&status);
    }
   return 0;
}
```

p2.c

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

int main(int argc, char **argv) {
    printf("Hello from p2, process id= () %d\n", getpid());
    printf("The arguments are %s %s %s\n",argv[0],argv[1],argv[2]);
    printf("Now, the child process will terminate\n");
    return 0;
}
```

compile&run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p1.c -o p1
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall p2.c -o p2
(base) mahmutunan@MacBook-Pro lecture18 % ./p1
Hello from p1, process id= () 30922
Hello from p2, process id= () 30923
The arguments are Hello CS 332
Now, the child process will terminate
(base) mahmutunan@MacBook-Pro lecture18 %
```

forkexecl.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    pid = fork();
```

```c
    if (pid == 0) { /* this is child process */
        execl( path: "/usr/bin/uname",  arg0: "uname", "-a", (char *)NULL);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execl");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecl.c -o exercise1
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise1
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
 2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[1290]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

- Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):
- functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer – *(char *)NULL (execl, execlp, and execle)*
- functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

- Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (*/*), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.
- Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

forkexecv.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;
    char *args[] = {"uname", "-a", (char *)NULL};

    pid = fork();
    if (pid == 0) { /* this is child process */
        execv( path: "/usr/bin/uname", args);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execv");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
```

forkexecv.c

```
18      } else if (pid > 0) { /* this is the parent process */
19          printf("Wait for the child process to terminate\n");
20          wait(&status); /* wait for the child process to terminate */
21          if (WIFEXITED(status)) { /* child process terminated normally */
22              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23          } else { /* child process did not terminate normally */
24              printf("Child process did not terminate normally!\n");
25              /* look at the man page for wait (man 2 wait) to determine
26                 how the child process was terminated */
27          }
28      } else { /* we have an error */
29          perror("fork"); /* use perror to print the system error message */
30          exit(EXIT_FAILURE);
31      }
32
33      printf("[%ld]: Exiting program .....\n", (long)getpid());
34
35      return 0;
36  }
37
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecv.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise2
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
 2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[30193]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

In order to see the difference between execl and execv, here is a line of code executing a

`ls -l -R -a`

with execl :

```
execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

with execv :

```
char* arr[] = {"ls", "-l", "-R", "-a", NULL};
execv("/bin/ls", arr);
```

The array of char* sent to execv will be passed to /bin/ls as argv (in `int main(int argc, char **argv)`)

https://stackoverflow.com/questions/32142164/how-to-use-execv-system-call-in-linux

forkexecvp.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp( file: argv[1],  argv: &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
        perror("execvp");
        exit(-1);
```

forkexecvp.c

```
22      } else if (pid > 0) { /* this is the parent process */
23          printf("Wait for the child process to terminate\n");
24          wait(&status); /* wait for the child process to terminate */
25          if (WIFEXITED(status)) { /* child process terminated normally */
26              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
27          } else { /* child process did not terminate normally */
28              printf("Child process did not terminate normally!\n");
29              /* look at the man page for wait (man 2 wait) to determine
30                 how the child process was terminated */
31          }
32      } else { /* we have an error */
33          perror("fork"); /* use perror to print the system error message */
34          exit(EXIT_FAILURE);
35      }
36
37      printf("[%ld]: Exiting program .....\n", (long)getpid());
38
39      return 0;
40  }
41
```

hello.c

```c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello from the execvp()\n");

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecvp.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall hello.c -o hello
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise3 ./hello
Wait for the child process to terminate
Hello from the execvp()
Child process exited with status = 0
[30387]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 % _
```

forkexecvp2.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp( file: argv[1], argv: &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execvp");
    exit(-1);
```

```c
    } else if (pid > 0) { /* this is the parent process */
        char name[BUFSIZ];

        printf("[%d]: Please enter your name: ", getpid());
        scanf("%s", name);
        printf("[%d-stdout]: Hello %s!\n", getpid(), name);
        fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);

        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}

```

- Here is the C APIs for the exec family of system calls:

#include <unistd.h>

```
int execl(const char *pathname, const char
*arg, ...);
int execlp(const char *filename, const char
*arg, ...);
int execle(const char *pathname, const char
*arg, ..., char * const envp[]);
int execv(const char *pathname, char *const
argv[]);
int execvp(const char *filename, char *const
argv[]);
int execvpe(const char *filename, char *const
argv[], char *const envp[]);
```

ls -lh

```
(base) mahmutunan@MacBook-Pro lecture19 % ls -lh
total 240
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   341B Oct  9 13:13 execle.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   326B Oct  9 13:19 execve.c
-rw-r--r--@ 1 mahmutunan  staff   198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:02 exercise4
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:13 exercise5
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:19 exercise6
```

execl

```c
int main(void) {
    char *binaryPath = "/bin/ls";
    char *arg0="ls";
    char *arg1 = "-lh";
    char *arg2 = "/Users/mahmutunan/Desktop/lecture19";

    execl(binaryPath, arg0, arg1, arg2, NULL);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execl.c -o exercise1
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise1
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   220B Oct  9 12:33 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   192B Oct  9 12:29 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:33 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:34 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:37 exercise4
(base) mahmutunan@MacBook-Pro lecture19 %
```

execlp

```c
#include <unistd.h>

int main(void) {
    char *commandName = "ls";
    char *arg1 = "-lh";
    char *arg2 = "/Users/mahmutunan/Desktop/lecture19";

    execlp(commandName,commandName, arg1, arg2, NULL);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execlp.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise2
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   192B Oct  9 12:29 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:34 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:37 exercise4
(base) mahmutunan@MacBook-Pro lecture19 %
```

execv

```c
#include <unistd.h>

int main(void) {
    char *binaryPath = "/bin/ls";
    char *args[]= {"ls","-lh","/Users/mahmutunan/Desktop/lecture19",NULL};

    execv(binaryPath,args);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execv.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise3
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   196B Oct  9 12:37 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:37 exercise4
(base) mahmutunan@MacBook-Pro lecture19 % _
```

execvp

```c
#include <unistd.h>

int main(void) {
    char *commandName= "ls";
    char *args[]= {commandName,"-lh","/Users/mahmutunan/Desktop/lecture19",NULL};

    execvp(commandName,args);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execvp.c -o exercise4
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise4
total 160
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:02 exercise4
(base) mahmutunan@MacBook-Pro lecture19 %
```

Environment / Environment Variable

- An important Unix concept is the **environment**, which is defined by environment variables. Some are set by the system, others by you, yet others by the shell, or any program that loads another program.
- A variable is a character string to which we assign a value. The value assigned could be a number, text, filename, device, or any other type of data.
- When you log in to the system, the shell undergoes a phase called **initialization** to set up the environment. Environment variables allow you to customize how the system works and the behavior of the applications on the system.

https://www.tutorialspoint.com/unix/unix-environment.htm

Environment / Environment Variable

Sr.No.	Variable & Description
1	**DISPLAY** Contains the identifier for the display that **X11** programs should use by default.
2	**HOME** Indicates the home directory of the current user: the default argument for the cd **built-in** command.
3	**IFS** Indicates the **Internal Field Separator** that is used by the parser for word splitting after expansion.
4	**LANG** LANG expands to the default system locale; LC_ALL can be used to override this. For example, if its value is **pt_BR**, then the language is set to (Brazilian) Portuguese and the locale to Brazil.
5	**LD_LIBRARY_PATH** A Unix system with a dynamic linker, contains a colonseparated list of directories that the dynamic linker should search for shared objects when building a process image after exec, before searching in any other directories.
6	**PATH** Indicates the search path for commands. It is a colon-separated list of directories in which the shell looks for commands.
7	**PWD** Indicates the current working directory as set by the cd command.
8	**RANDOM** Generates a random integer between 0 and 32,767 each time it is referenced.

https://www.tutorialspoint.com/unix/unix-environment.htm

execle

```c
#include <unistd.h>

int main(void) {
    char *binaryPath= "/bin/bash";
    char *arg1 = "-c";
    char *arg2 = "echo \"Visit $HOSTNAME:$PORT from your browser.\"";
    char *const env[] = {"HOSTNAME=https://www.uab.edu/cas/computerscience/", "PORT=8080", NULL};

    execle(binaryPath, binaryPath,arg1, arg2, NULL, env);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execle.c -o exercise5
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise5
Visit https://www.uab.edu/cas/computerscience/:8080 from your browser.
(base) mahmutunan@MacBook-Pro lecture19 % _
```

execve

```c
#include <unistd.h>

int main(void) {
    char *binaryPath= "/bin/bash";
    char *const args[] = {binaryPath,"-c","echo \"Visit $HOSTNAME:$PORT from your browser.\"",NULL};
    char *const env[] = {"HOSTNAME=https://www.uab.edu/cas/computerscience/", "PORT=8080", NULL};

    execve(binaryPath, args, env);

    return 0;
}
```

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc execve.c -o exercise6
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise6
Visit https://www.uab.edu/cas/computerscience/:8080 from your browser.
(base) mahmutunan@MacBook-Pro lecture19 %
```

Ref : https://linuxhint.com/exec_linux_system_call_c/

forkexecl.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    pid = fork();
```

```c
    if (pid == 0) { /* this is child process */
        execl( path: "/usr/bin/uname",  arg0: "uname", "-a", (char *)NULL);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execl");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecl.c -o exercise1
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise1
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
 2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[1290]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

- Let us look at the different versions of the exec functions. There are two classes of exec functions based on whether the argument is a list of separate values (l versions) or the argument is a vector (v versions):
- functions that take a variable number of command-lines arguments each as an array of characters terminated with a null character and the last argument is a null pointer – *(char *)NULL (execl, execlp, and execle)*
- functions that take the command-line arguments as a pointer to an array of pointers to the arguments, similar to argv parameter used by the main method (execv, execvp, and execvpe)

- Functions that have *p* in the name use *filename* as the first argument while functions without *p* use the *pathname* as the first argument. If the filename contains a slash character (*/*), it is considered as a pathname, otherwise, all directories specified by the PATH environment variable are searched for the executable.
- Functions that end in *e* have an additional argument – a pointer to an array of pointers to the environment strings.

forkexecv.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;
    char *args[] = {"uname", "-a", (char *)NULL};

    pid = fork();
    if (pid == 0) { /* this is child process */
        execv( path: "/usr/bin/uname", args);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execv");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
```

forkexecv.c

```
18      } else if (pid > 0) { /* this is the parent process */
19          printf("Wait for the child process to terminate\n");
20          wait(&status); /* wait for the child process to terminate */
21          if (WIFEXITED(status)) { /* child process terminated normally */
22              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
23          } else { /* child process did not terminate normally */
24              printf("Child process did not terminate normally!\n");
25              /* look at the man page for wait (man 2 wait) to determine
26                 how the child process was terminated */
27          }
28      } else { /* we have an error */
29          perror("fork"); /* use perror to print the system error message */
30          exit(EXIT_FAILURE);
31      }
32
33      printf("[%ld]: Exiting program .....\n", (long)getpid());
34
35      return 0;
36  }
37
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecv.c -o exercise2
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise2
Wait for the child process to terminate
Darwin MacBook-Pro.local 19.6.0 Darwin Kernel Version 19.6.0: Thu Jun 18 20:49:00 PDT
 2020; root:xnu-6153.141.1~1/RELEASE_X86_64 x86_64
Child process exited with status = 0
[30193]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

In order to see the difference between execl and execv, here is a line of code executing a

`ls -l -R -a`

with execl :

```
execl("/bin/ls", "ls", "-l", "-R", "-a", NULL);
```

with execv :

```
char* arr[] = {"ls", "-l", "-R", "-a", NULL};
execv("/bin/ls", arr);
```

The array of char* sent to execv will be passed to /bin/ls as argv (in `int main(int argc, char **argv)`)

https://stackoverflow.com/questions/32142164/how-to-use-execv-system-call-in-linux

forkexecvp.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp( file: argv[1],  argv: &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
        perror("execvp");
        exit(-1);
```

forkexecvp.c

```
22      } else if (pid > 0) { /* this is the parent process */
23          printf("Wait for the child process to terminate\n");
24          wait(&status); /* wait for the child process to terminate */
25          if (WIFEXITED(status)) { /* child process terminated normally */
26              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
27          } else { /* child process did not terminate normally */
28              printf("Child process did not terminate normally!\n");
29              /* look at the man page for wait (man 2 wait) to determine
30                 how the child process was terminated */
31          }
32      } else { /* we have an error */
33          perror("fork"); /* use perror to print the system error message */
34          exit(EXIT_FAILURE);
35      }
36
37      printf("[%ld]: Exiting program .....\n", (long)getpid());
38
39      return 0;
40  }
41
```

hello.c

```c
#include <stdio.h>
int main(int argc, char **argv) {
    printf("Hello from the execvp()\n");

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall forkexecvp.c -o exercise3
(base) mahmutunan@MacBook-Pro lecture18 % gcc -Wall hello.c -o hello
(base) mahmutunan@MacBook-Pro lecture18 % ./exercise3 ./hello
Wait for the child process to terminate
Hello from the execvp()
Child process exited with status = 0
[30387]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture18 %
```

forkexecvp2.c

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

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp( file: argv[1], argv: &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execvp");
    exit(-1);
```

```c
    } else if (pid > 0) { /* this is the parent process */
        char name[BUFSIZ];

        printf("[%d]: Please enter your name: ", getpid());
        scanf("%s", name);
        printf("[%d-stdout]: Hello %s!\n", getpid(), name);
        fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);

        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}

```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture19 % gcc -Wall forkexecvp2.c -o exercise7
(base) mahmutunan@MacBook-Pro lecture19 % ./exercise7 ls -lh
[65577]: Please enter your name: total 280
-rw-r--r--@ 1 mahmutunan  staff   239B Oct  9 12:56 execl.c
-rw-r--r--@ 1 mahmutunan  staff   341B Oct  9 13:13 execle.c
-rw-r--r--@ 1 mahmutunan  staff   221B Oct  9 12:58 execlp.c
-rw-r--r--@ 1 mahmutunan  staff   194B Oct  9 12:59 execv.c
-rw-r--r--@ 1 mahmutunan  staff   326B Oct  9 13:19 execve.c
-rw-r--r--@ 1 mahmutunan  staff   198B Oct  9 13:01 execvp.c
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:56 exercise1
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 12:58 exercise2
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:00 exercise3
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:02 exercise4
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:13 exercise5
-rwxr-xr-x  1 mahmutunan  staff    12K Oct  9 13:19 exercise6
-rwxr-xr-x  1 mahmutunan  staff    13K Oct  9 13:51 exercise7
-rw-r--r--@ 1 mahmutunan  staff   1.8K Oct  5 19:48 forkexecvp2.c
mahmut
[65577-stdout]: Hello mahmut!
[65577-stderr]: Hello mahmut!
Child process exited with status = 0
[65577]: Exiting program .....
(base) mahmutunan@MacBook-Pro lecture19 %
```

Linux

- Linux and Unix are different, but they do have a relationship with each other
  - Linux is derived from Unix.
- Linux is just the kernel and not the complete OS. This Linux kernel is generally packaged in Linux distributions which thereby makes it a complete OS
- There are many different versions of Linux, and the core of Linux is free to distribute and use.
- Normally, distributions are made for specific reasons and have been tailored to address a series of concerns

https://www.softwaretestinghelp.com/unix-vs-linux/#:~:text=Linux%20refers%20to%20the%20kernel,family%20of%20derived%20operating%20systems.
https://www.linux.org/forums/linux-beginner-tutorials.123/

Processes in a Linux environment

- User processes in a Linux environment could be in one of the following three states: `foreground, background`, or `suspended.`
- Most interactive applications that take input from the keyboard or command-line argument and display output in a terminal are considered as foreground processes. Till now, we have been executed all our programs as foreground processes by typing the name of the command in the bash shell.

- Non-interactive processes that are typically not connected to a terminal and execute in the background are considered as background processes.
- You can execute a program in the background by typing the program name followed by the symbol `&` at the end.
- You will notice that the shell returns the command-prompt with the background process number and the corresponding process identifier (PID) of the process that was created.
- For example:

```
$ nano myprog.c &
[1] 16946
$
```

- You can display various jobs that are currently running in the background using the *jobs* command. It will show the job number, the current state of the job, and the job name. For example:

```
$ jobs
[1]+  Stopped                 nano myprog.c
$
```

- If you like to list additional information such as the PID of the job you can use the -l option. For example:

```
$ jobs -l
[1]+ 16946 Stopped (tty output)  nano myprog.c
$
```

In case of the above example, we have invoked an editor and it has been stopped since it requires terminal to display the output and continue. If we had created a non-interactive job in the background, then it would be in the running state. For example:

```
$ sleep 20 &
[2] 17513
$ jobs
[1]+  Stopped                 nano myprog.c
[2]-  Running                 sleep 20 &
$
```

- We can bring a job that is running in the background to foreground by using the *fg* command. For example, to switch the *sleep* process to foreground, we specify the job number after the % symbol:

```
$ fg %2
sleep 20
$ jobs
[1]+  Stopped                 nano myprog.c
$
```

- You will notice that the sleep process will return the command prompt in the terminal when it completes execution and when you type *jobs* again, it will only show one process. You can use `fg` command to switch to the editor using: `fg %1`.
- You can continue with your edits, save the file, and exit the editor.
- After you exit, if you type `jobs` again, you will notice that it does not list any processes since the *nano* process is no longer executing.
- If you like to suspend a foreground process then type *Control-Z* when the program is executing and that process will be suspended

- For example, if we started the sleep process in foreground and would like to suspend it, then type *Control-Z* and you will see a message similar to what you saw when you started a process in background:

```
$
$ jobs
$ sleep 100
^Z
[1]+  Stopped                 sleep 100
$ jobs
[1]+  Stopped                 sleep 100
$
```

- However, notice that the sleep process is stopped (it is not running) unlike the previous case when it was running in the background. If you like the sleep process to continue then you have to using the *bg* command as follows:

```
$ bg %1
[1]+ sleep 100 &
$ jobs
[1]+  Running                 sleep 100 &
$
[1]+  Done                    sleep 100
$ jobs
```

- Now you notice that the sleep process is running and when it is done you will see the message *Done* in your terminal.
- There are special background processes that are started at system startup and they continue to run till the system is shutdown.
- These special background processes are called ***daemons***.
- These processes typically end in “d” and some examples are: systemd, crond, ntpd, nfsd, sshd, httpd, named.
- If you like to terminate a process that is executing in the foreground, you use Control-C to kill it.
- If you like to terminate a process in the background, you could bring it to foreground and then use Control-C or use the *kill* command to terminate the background process directly

- For example:
- ```
  $ jobs
  $ sleep 100 &
  [1] 1519
  $ jobs
  [1]+  Running                 sleep 100 &
  $ kill %1
  [1]+  Terminated              sleep 100
  $
  $ jobs
  ```
  $ You can also provide the PID as the argument to *kill* command to terminate a process.
  ```

Monitor processes in Linux environment

- You can use the *ps* command to display information about various processes running on a Linux system. Login to one of CS Linux systems and enter the *ps* command, you will see the following information displayed:

```
~~{2.00}~{unan@vulcan18:~}~~
[$ ps
   PID TTY          TIME CMD
30544 pts/1    00:00:00 bash
30692 pts/1    00:00:00 ps
```

ps -man page

- ps - report a snapshot of the current processes.
- **ps** [*options*]

```
To see every process on the system using standard syntax:
   ps -e
   ps -ef
   ps -eF
   ps -ely

To see every process on the system using BSD syntax:
   ps ax
   ps axu

To print a process tree:
   ps -ejH
   ps axjf

To get info about threads:
   ps -eLf
   ps axms

To get security info:
   ps -eo euser,ruser,suser,fuser,f,comm,label
   ps axZ
   ps -eM
```

https://man7.org/linux/man-pages/man1/ps.1.html

- By default, *ps* lists processes for the current user that are associated with the terminal that invoked the command and the output is unsorted.
- The following information is shown above: the process ID (PID), the terminal associated with the process (TTY), the cumulative CPU time in hh:mm:ss format (TIME), and the executable name (CMD).
- You will notice that there are two processes currently executing – bash and ps (the command you just executed) along with their corresponding process ID (in the first column under PID).

- The *ps* command has a large number of options that you can use to get more detailed information about the various processes currently running. We will look at some of these options below, you can find out more about these options using *man ps*.
- The *-u username* option lists all processes that belong to the user *username*:

```
$ ps -u unan
  PID TTY          TIME CMD
30543 ?        00:00:00 sshd
30544 pts/1    00:00:00 bash
30716 pts/1    00:00:00 ps
```

- You can select all processes owned by you (runner of the **ps command**, root in this case), type:

```
ps -x
```

- We can also view every process running with **root** user privileges (real & effective ID) in user format.

```
ps -U root -u root
```

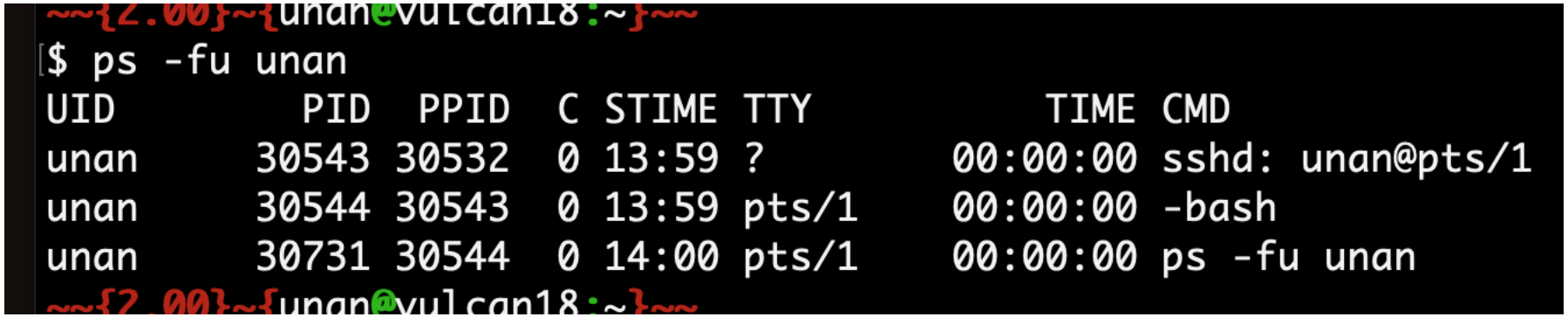
- The command below allows you to view the PID, PPID, username and command of a process.

```
$ ps -eo pid,ppid,user,cmd
```

- To select a specific process by its name, use the -C flag, this will also display all its child processes.

```
$ ps -C sshd
```

- Now you notice that there is an additional process (sshd) that belongs to you and there is no terminal associated with that process (hence the ? under the TTY column). This process was started by the OS when you connected to this computer using an SSH client. You can use the -f or -F option to get a full listing:



```
[$ ps -Fu unan
UID        PID  PPID  C    SZ   RSS PSR STIME TTY          TIME CMD
unan     30543 30532  0 49950  2784   0 13:59 ?        00:00:00 sshd: unan@pts/1
unan     30544 30543  0 33493  3748   0 13:59 pts/1    00:00:00 -bash
unan     30746 30544  0 42042  1940   0 14:00 pts/1    00:00:00 ps -Fu unan
```

Now we see several additional fields displayed such the user ID, parent PI, process with command-line options, etc.

By looking at the PID and PPID we can identify that the *ps* command was created by the *bash* process and the *bash* process was in turn created by the *sshd* process. We can display the process tree with the ps command using the --forest option:

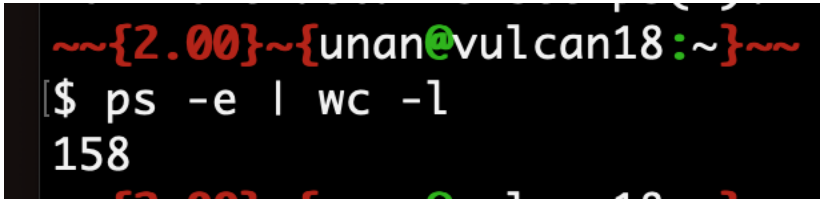
```
~~{2.00}~{unan@vulcan18:~}~~
[$ ps -fu unan --forest
UID          PID  PPID  C STIME TTY          TIME CMD
unan       30543 30532  0 13:59 ?        00:00:00 sshd: unan@pts/1
unan       30544 30543  0 13:59 pts/1    00:00:00  \_ -bash
unan       30789 30544  0 14:01 pts/1    00:00:00      \_ ps -fu unan --forest
```

You can use *ps* to list every process currently running (not just processes that belong to you) using the -e option

```
ps -e | more
ps -ef | more
ps -eF | more
ps -ely | more
```

You have to press the **spacebar** to scroll through the list.

- You can send the output to the *wc* command to determine the total number of processes currently running on the system, for example:



At this the above command was executed on one of the CS Linux system, we had 158 processes currently running on the system, even though there are only three processes that belong to you. What are all these processes? Who is running these processes?

- You can also use the ***pstree*** command to display the process tree (you can find out more about the various options supported by *pstree* using *man pstree*).
- For example:

```
pstree -np | more
```

```
$ ps -e | more
  PID TTY          TIME CMD
    1 ?        00:14:34 systemd
    2 ?        00:00:03 kthreadd
    4 ?        00:00:00 kworker/0:0H
    6 ?        00:00:29 ksoftirqd/0
    7 ?        00:00:05 migration/0
    8 ?        00:00:00 rcu_bh
    9 ?        00:10:59 rcu_sched
   10 ?        00:00:00 lru-add-drain
   11 ?        00:00:34 watchdog/0
   12 ?        00:00:29 watchdog/1
   13 ?        00:00:06 migration/1
   14 ?        00:00:23 ksoftirqd/1
```

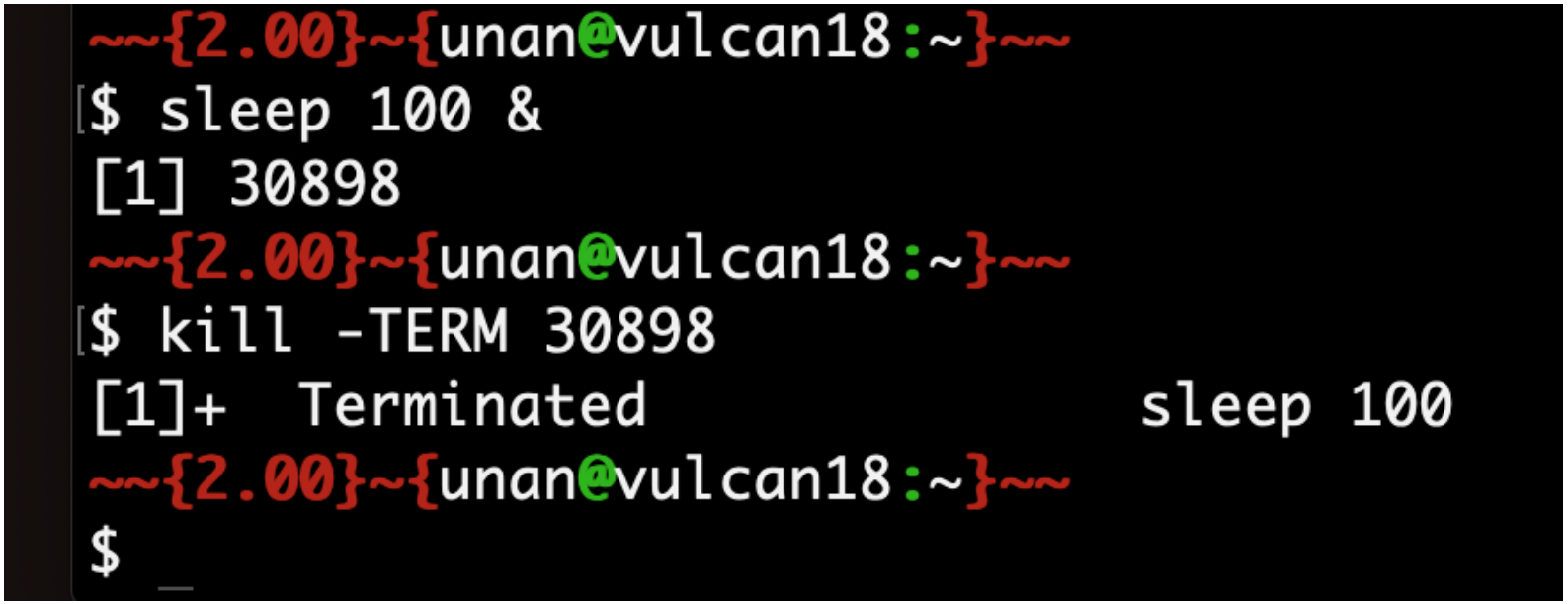
- Similarly, you can use the `top` command (instead of *ps*) to display all processes currently running (you have to enter *q* to quit *top*).
- The *top* command provides a real-time update on the various processes running on the system.
- You can look at processes that belong to a specific user by typing *u* and the user name when top is running or start *top* with the `-u user` option.
- It provides a dynamic real-time view of the running system. Usually, this command shows the summary information of the system and the list of processes or threads which are currently managed by the Linux Kernel.
- https://man7.org/linux/man-pages/man1/top.1.html

kill

- To terminate a process, we can use the *kill* command. The *kill* command can take the PID or the command name and terminate a specific process with the given PID or terminate all processes with the specified command name. We can also specify the type of signal, either as a signal name (*e.g.,* KILL for kill) or signal number (9 for kill), to send to a process with the *kill* command. Here is an example that shows how to use the *kill* command to terminate a process using PID.

kill - man page

- kill - terminate a process
- The command **kill** sends the specified *signal* to the specified processes or process groups.
- If no signal is specified, the TERM signal is sent. The default action for this signal is to terminate the process. This signal should be used in preference to the KILL signal (number 9), since a process may install a handler for the TERM signal in order to perform clean-up steps before terminating in an orderly fashion
- https://man7.org/linux/man-pages/man1/kill.1.html



You can find the complete list of signal names and numbers using t he *-l* option of *kill* command.
We will discuss more about signals in the later labs

```
kill SIGNAL PID
```

- Where SIGNAL is the signal to be sent and PID is the Process ID to be killed.
- Let's say we need to terminate the process 3827. We need to send the kill signal;

```
kill -9 3827
```

To display all the available signals, we should use below command option

`kill -l`

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL       5) SIGTRAP
 6) SIGABRT      7) SIGBUS       8) SIGFPE       9) SIGKILL     10) SIGUSR1
11) SIGSEGV     12) SIGUSR2     13) SIGPIPE     14) SIGALRM     15) SIGTERM
16) SIGSTKFLT   17) SIGCHLD     18) SIGCONT     19) SIGSTOP     20) SIGTSTP
21) SIGTTIN     22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO       30) SIGPWR
31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1  36) SIGRTMIN+2  37) SIGRTMIN+3
38) SIGRTMIN+4  39) SIGRTMIN+5  40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8
43) SIGRTMIN+9  44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9  56) SIGRTMAX-8  57) SIGRTMAX-7
58) SIGRTMAX-6  59) SIGRTMAX-5  60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2
63) SIGRTMAX-1  64) SIGRTMAX
```

Explore the /proc file system

- The proc file system (procfs) is a virtual file system that is created by the OS at system boot time to provide an interface between the kernel space and user space.
- It is commonly mounted at /proc.
- It provides information of processes currently running on the system and tools such as *ps* use this to display information about these processes

ls -l /proc

```
~~{1.06}~{unan@vulcan16:~}~~
$ ls -l /proc
total 0
dr-xr-xr-x  9 root          root                0 Apr 28 14:54 1
dr-xr-xr-x  9 root          root                0 Sep 23 22:27 10
dr-xr-xr-x  9 root          root                0 Oct  1 21:17 10596
dr-xr-xr-x  9 root          root                0 Oct  1 21:17 10597
dr-xr-xr-x  9 root          root                0 Sep 23 22:27 11
dr-xr-xr-x  9 root          root                0 Sep 23 22:27 111
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1175
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1176
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1177
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1179
dr-xr-xr-x  9 root          root                0 Sep 23 22:27 12
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1218
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1296
dr-xr-xr-x  9 postfix       postfix             0 Oct  1 21:16 1298
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1299
dr-xr-xr-x  9 root          root                0 Sep 23 22:27 13
dr-xr-xr-x  9 root          root                0 Oct  1 21:16 1313
```

File	Description
/proc/cpuinfo	information about the CPU architecture, used by *lscpu* command
/proc/loadavg	system load averages data, used by *uptime* command
/proc/meminfo	memory usage statistics on the system, used by *free* command
/proc/stat	kernel/system statistics
/proc/version	version of the kernel currently running on the system, used by *uname*command
/proc/[pid]	a subdirectory for each running process
/proc/[pid]/cmdline	command string of the process along with arguments separated by null (‘\0’) character
/proc/[pid]/cmd	a symbolic link to the current working directory of the process
/proc/[pid]/environ	environment variables and values used by the process separated by null (‘\0’) character (use strings /proc/[pid]/environ to display the environment variable and values)
/proc/[pid]/maps	information on memory mapped regions of the process
/proc/[pid]/mem	information to access pages of the process through I/O calls
/proc/[pid]/stat	status information about the process, used by *ps* command
/proc/[pid]/statm	status of memory used by the process, measured in pages

- Let’s take a look at one of the files :

```
# cat /proc/meminfo
```

```
~~{    }~{unan@vulcan0:~}~~
$ cat /proc/meminfo
MemTotal:        3880088 kB
MemFree:         2366368 kB
MemAvailable:    3141080 kB
Buffers:               0 kB
Cached:           841836 kB
SwapCached:         7264 kB
Active:           687940 kB
Inactive:         181940 kB
Active(anon):      93296 kB
Inactive(anon):   113528 kB
Active(file):     594644 kB
Inactive(file):    68412 kB
Unevictable:         120 kB
Mlocked:             120 kB
SwapTotal:       6160380 kB
SwapFree:        6083580 kB
Dirty:               404 kB
Writeback:             0 kB
AnonPages:         25520 kB
```

- You can list the contents of /proc using the *ls* command and view files using the *cat* command.
- For files that contain strings separated with null characters you have use the *strings* command to display the contents of such files correctly.
- Here is an example to look at the environ file for the *bash* process:

```
$ strings /proc/$$/environ
```

- Note: $$ in bash refers to the process ID of the current process, you can replace it with the actual PID of bash and test the above command.

Processes and Threads

- The unit of dispatching is referred to as a ***thread*** or ***lightweight process***
- The unit of resource ownership is referred to as a ***process*** or ***task***
- ***Multithreading -*** The ability of an OS to support multiple, concurrent paths of execution within a single process

Threads

- Each thread belongs to exactly one process and no thread can exist outside a process.
- Each thread represents a separate flow of control.
- Threads have been successfully used in implementing network servers and web server.
- They also provide a suitable foundation for parallel execution of applications on shared memory multiprocessors.

S.N.	Process	Thread
1	Process is heavy weight or resource intensive.	Thread is light weight, taking lesser resources than a process.
2	Process switching needs interaction with operating system.	Thread switching does not need to interact with operating system.
3	In multiple processing environments, each process executes the same code but has its own memory and file resources.	All threads can share same set of open files, child processes.
4	If one process is blocked, then no other process can execute until the first process is unblocked.	While one thread is blocked and waiting, a second thread in the same task can run.
5	Multiple processes without using threads use more resources.	Multiple threaded processes use fewer resources.
6	In multiple processes each process operates independently of the others.	One thread can read, write or change another thread's data.

https://www.tutorialspoint.com/operating_system/os_multi_threading.htm

Single Threaded Approaches

- A single thread of execution per process, in which the concept of a thread is not recognized, is referred to as a single-threaded approach
- MS-DOS is an example

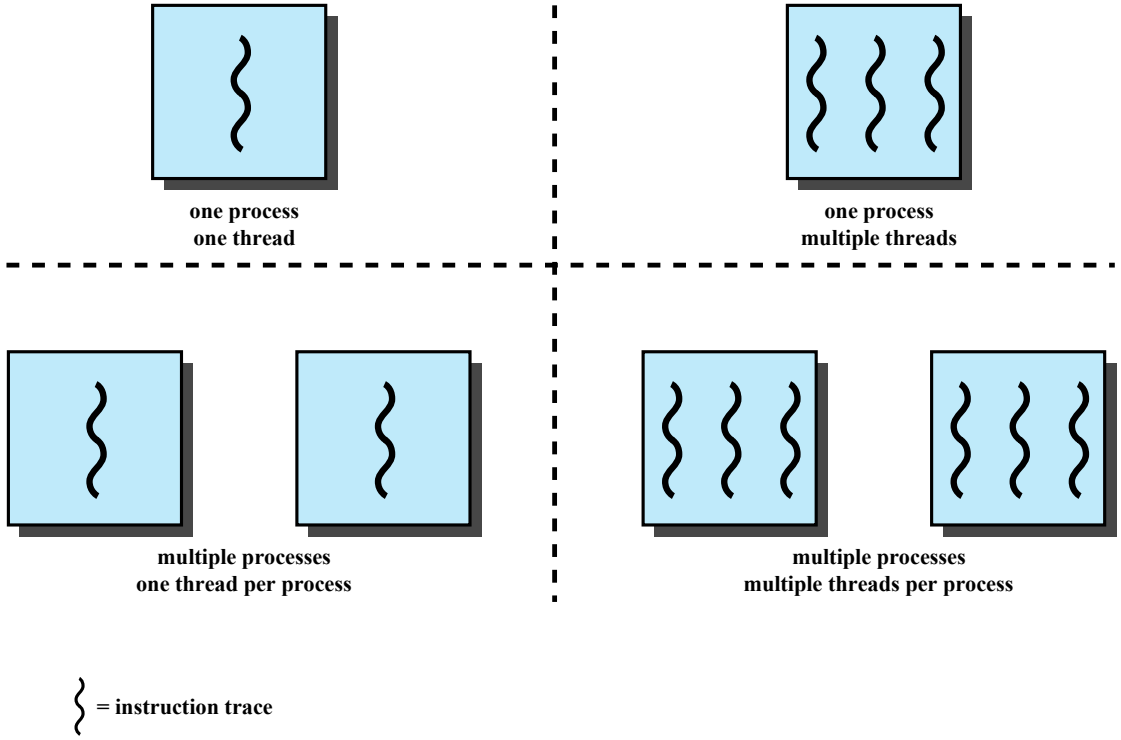


Figure 4.1 Threads and Processes

Multithreaded Approaches

- The right half of the figure depicts multithreaded approaches
- A Java run-time environment is an example of a system of one process with multiple threads

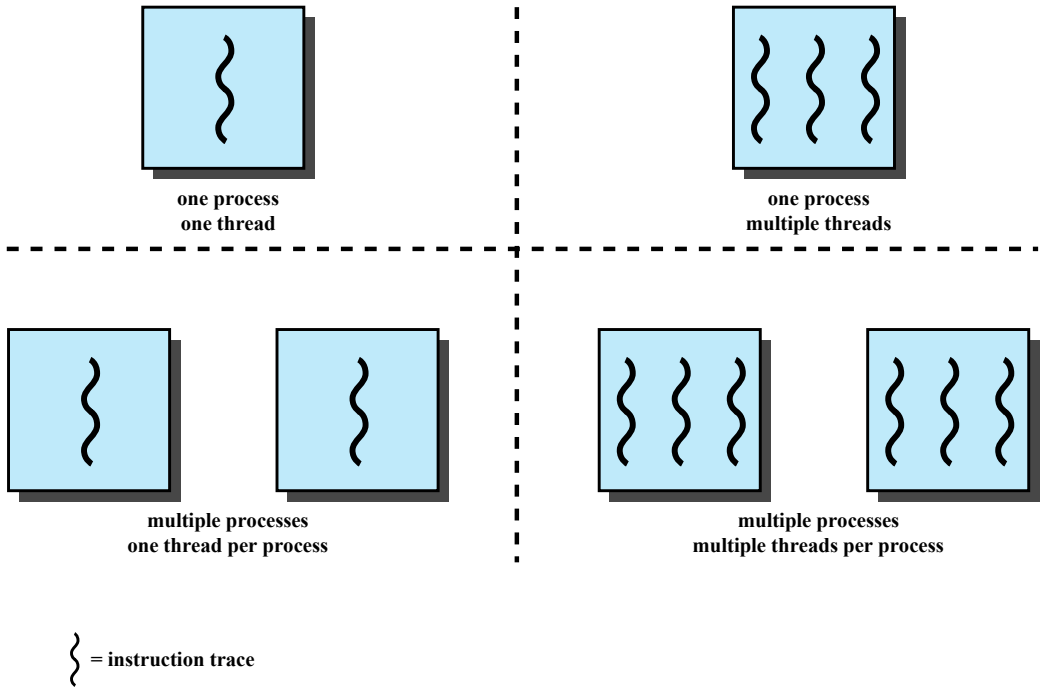
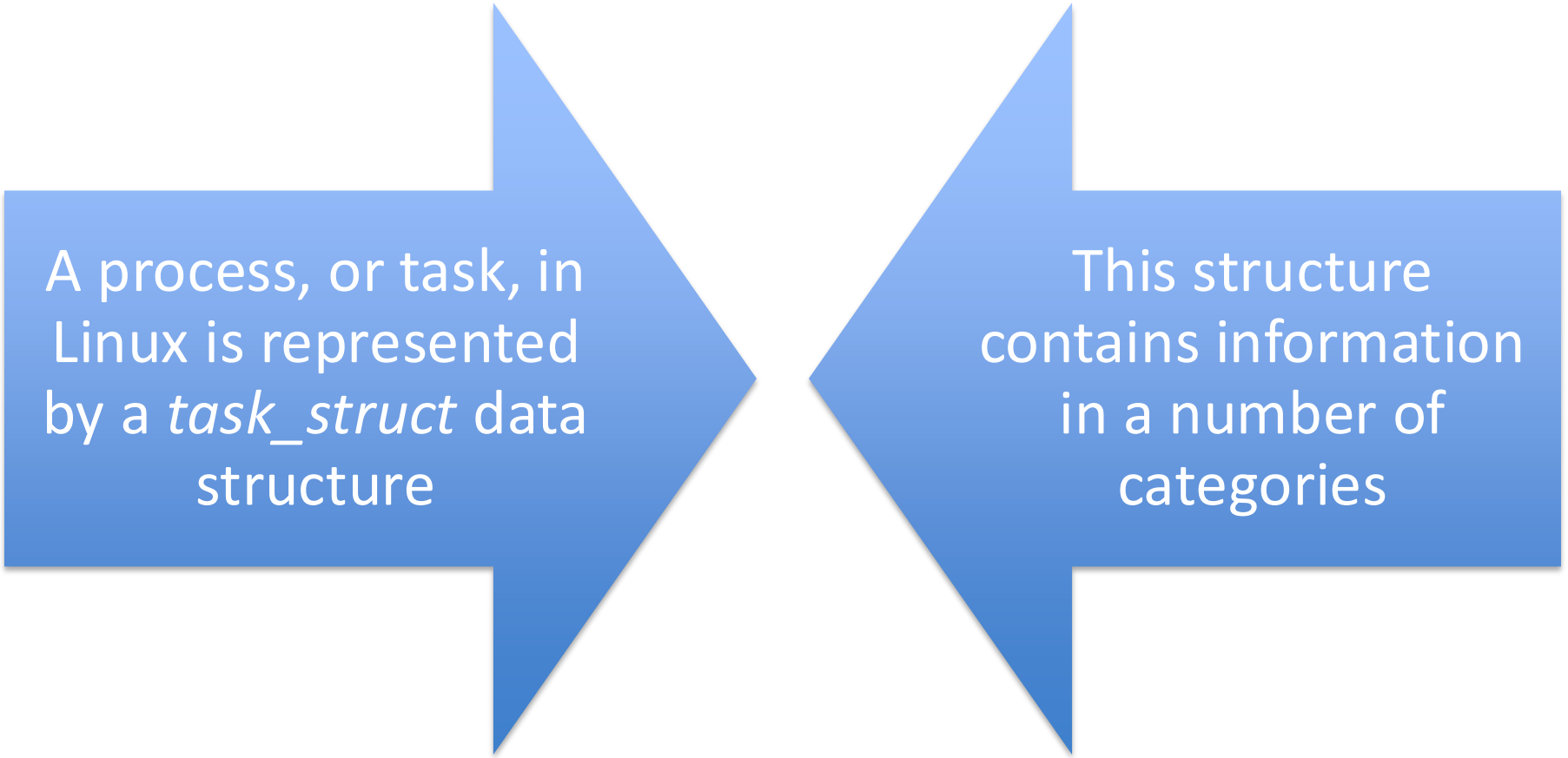


Figure 4.1 Threads and Processes

Linux Tasks



• State, • Scheduling information, • Identifiers, • Interprocess communication, • Links, • Times and timers, • File system, • Address space, • Processor-specific context:

Concurrency

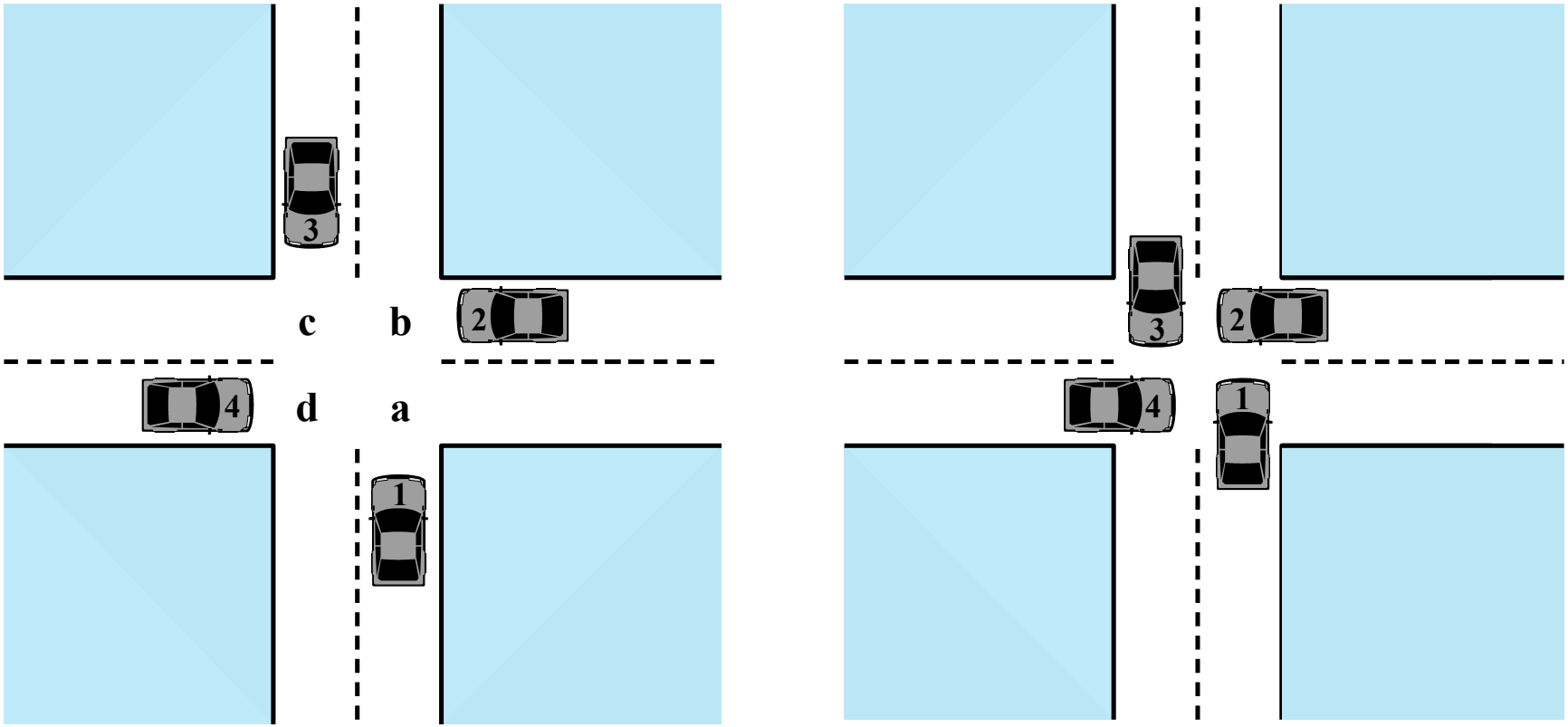
- Fundamental to all of these areas, and fundamental to OS design, is **concurrency**.
- Concurrency encompasses a host of design issues, including communication among processes, sharing of and competing for resources (such as memory, files, and I/O access), synchronization of the activities of multiple processes, and allocation of processor time to processes.
- We shall see that these issues arise not just in multiprocessing and distributed processing environments but even in single-processor multiprogramming systems.

Race Condition

- Occurs when multiple processes or threads read and write data items
- The final result depends on the order of execution
  - The “loser” of the race is the process that updates last and will determine the final value of the variable

Deadlock

- **The *permanent* blocking of a set of processes that either compete for system resources or communicate with each other**
- A set of processes is deadlocked when each process in the set is blocked awaiting an event that can only be triggered by another blocked process in the set
- Permanent because none of the events is ever triggered
- No efficient solution in the general case



(a) Deadlock possible

(b) Deadlock

Figure 6.1 Illustration of Deadlock

Deadlock Approaches

- There is no single effective strategy that can deal with all types of deadlock
- Three approaches are common:

- **Deadlock prevention**
  - Disallow one of the three necessary conditions for deadlock occurrence, or prevent circular wait condition from happening

- **Deadlock avoidance**
  - Do not grant a resource request if this allocation might lead to deadlock

- **Deadlock detection**
  - Grant resource requests when possible, but periodically check for the presence of deadlock and take action to recover

UNIX Concurrency Mechanisms

- UNIX provides a variety of mechanisms for interprocessor communication and synchronization including:

Signals

- A software mechanism that informs a process of the occurrence of asynchronous events
    - Similar to a hardware interrupt, but does not employ priorities
- A signal is delivered by updating a field in the process table for the process to which the signal is being sent
- A process may respond to a signal by:
    - Performing some default action
    - Executing a signal-handler function
    - Ignoring the signal

Signals

- Signals are software interrupts that provide a mechanism to deal with asynchronous events (e.g., a user pressing Control-C to interrupt a program that the shell is currently executing).
- A signal is a notification to a process that an event has occurred that requires the attention of the process (this typically interrupts the normal flow of execution of the interrupted process).
- Signals can also be used as a synchronization technique or even as a simple form of interprocess communication.
- Signals could be generated by hardware interrupts, the program itself, other programs, the OS kernel, or by the user.

- All these signals have a unique symbolic name and starts with SIG. The standard signals (also called POSIX reliable signals) are defined in the header file *<signal.h>*.
- Here are some examples:
- `SIGINT`: Interrupt a process from keyboard (e.g., pressing Control-C). The process is terminated.
- `SIGQUIT`: Interrupt a process to quit from keyboard (e.g., pressing Control-/). The process is terminated and a core file is generated.
- `SIGTSTP`: Interrupt a process to stop from keyboard (e.g., pressing Control-Z). The process is stopped from executing.
- `SIGUSR1` and `SIGUSR2`: These are user-defined signals, for use in application programs.

- ***NOTE***: Please see Section 10.2 and Figure 10.1 in the textbook for a complete list of UNIX System signals. You can also find more details using *man 7 signal* on any CS UNIX system (*man signal* on Mac).
- After a signal is generated, it is delivered to a process to perform some action in response to this signal.
- Since signals are asynchronous events, a process has to decide ahead of time how to respond when the particular signal is delivered.
- There are three different options possible when a signal is delivered to a process:

- Perform the default action. Most signals have a default action associated with them, if the process does not change the default behavior then the default action specific to that particular signal will occur.
  - The predefined default signal is specified as `SIG_DFL`.
- Ignore the signal. If a signal is ignored, then the default action is performed.
  - The predefined ignore signal handler is specified as `SIG_IGN`.
- Catch and Handle the signal. It is also possible to override the default action and invoke a specific user-defined task when a signal is received.
  - This is usually performed by invoking a signal handler using *`signal()`* or *`sigaction()`* system calls.
- However, the signals `SIGKILL` and `SIGSTOP` cannot be caught, blocked, or ignored.

signal()

- signal - ANSI C signal handling
-
```
typedef void (*sighandler_t)(int);
```
- `sighandler_t signal(int signum, sighandler_t handler);`

- **signal**() sets the disposition of the signal *signum* to *handler*, which is either **SIG_IGN**, **SIG_DFL**, or the address of a programmer-defined function (a "signal handler").

Sending Signals Using The Keyboard

- Ctrl-C to send an INT signal (SIGINT) to the running process.
  - This signal causes the process to immediately terminate.
- Ctrl-Z to send a TSTP signal (SIGTSTP) to the running process.
  - This signal causes the process to suspend execution.
- Ctrl-\ to send a ABRT signal (SIGABRT) to the running process.
  - This signal causes the process to immediately terminate.
  - Ctrl-\ doing the same as Ctrl-C but it gives us some better flexibility.

infinite loop

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

void handleSignINT(int sig)
{
    printf("the signal caught =  %d\n", sig);
}

int main()
{
    signal(SIGINT, handleSignINT);
    while (1==1)
    {
        printf("Hello CS332 \n");
        sleep(1);
    }
    return 0;
}

```

compile & run

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc helloLoop.c -o helloLoop
(base) mahmutunan@MacBook-Pro lecture23 % ./helloLoop
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
Hello CS332
^Cthe signal caught =  2
Hello CS332
Hello CS332
^\zsh: quit       ./helloLoop
```

- Now, let's write a program to handle a simple signal that catches either of the two user-defined signals and prints the signal number (see Figure 10.2 in the textbook).
- 1. Create a user-defined function (signal handler) that will be invoked when a signal is received:

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

static void sig_usr(int signo) {
    if (signo == SIGUSR1) {
        printf("received SIGUSR1\n");
    } else if (signo == SIGUSR2) {
        printf("received SIGUSR2\n");
    } else {
        printf("received signal %d\n", signo);
    }
}

```

- 2. Now let’s call above user-defined signal.

```
16  int main(void) {
17      if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
18          printf("can't catch SIGUSR1\n");
19          exit(-1);
20      }
21      if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
22          printf("can't catch SIGUSR2\n");
23          exit(-1);
24      }
25      for ( ; ; )
26          pause();
27
28      return 0;
29  }
```

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigusr.c -o sigusr
(base) mahmutunan@MacBook-Pro lecture23 % ls
sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % ./sigusr &
[1] 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
```

```
(base) mahmutunan@MacBook-Pro lecture23 % kill -SIGUSR1 8122
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -STOP 8122
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + suspended (signal)  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR1 %1
received SIGUSR1
(base) mahmutunan@MacBook-Pro lecture23 % kill -USR2 %1
received SIGUSR2
(base) mahmutunan@MacBook-Pro lecture23 % kill -CONT 8122
(base) mahmutunan@MacBook-Pro lecture23 % jobs
[1]  + running    ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % kill -TERM %1
[1]  + terminated  ./sigusr
(base) mahmutunan@MacBook-Pro lecture23 % jobs
(base) mahmutunan@MacBook-Pro lecture23 % _
```

- In the example we used the *kill* command that we used in the previous lecture to generate the signal.
- While we used the kill command in the previous lecture to terminate a process using the TERM signal (the default signal if no signal is specified), the kill command could be used to generate any other signal that is supported by the kernel.

- You can list all signals that can be generated using *kill -l*. For example, on the CS Linux systems you see the following output:

```
$ kill -l
1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP
6) SIGABRT 7) SIGBUS 8) SIGFPE 9) SIGKILL 10) SIGUSR1
11) SIGSEGV 12) SIGUSR2 13) SIGPIPE 14) SIGALRM 15) SIGTERM
16) SIGSTKFLT 17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP
21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU 25) SIGXFSZ
26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGIO 30) SIGPWR
31) SIGSYS 34) SIGRTMIN 35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3
38) SIGRTMIN+4 39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8
43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13 52) SIGRTMAX-12
53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7
58) SIGRTMAX-6 59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2
63) SIGRTMAX-1 64) SIGRTMAX
```

Exercise 2

- We will now extend the exercise 1 to handle other signal generated from keyboard such as Control-C (SIGINT), Control-Z (SIGTSTP), and Control-\ (SIGQUIT). In this example we will use a single signal handler to handle all these signals using a switch statement (instead of separate signal handlers).

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

static void sig_usr(int signo) {
    switch(signo) {
        case SIGINT:
         printf("received SIGINT signal %d\n", signo);
         break;
        case SIGQUIT:
         printf("received SIGQUIT signal %d\n", signo);
         break;
        case SIGUSR1:
         printf("received SIGUSR1 signal %d\n", signo);
         break;
        case SIGUSR2:
         printf("received SIGUSR2 signal %d\n", signo);
         break;
        case SIGTSTP:
         printf("received SIGTSTP signal %d\n", signo);
         break;
        default:
         printf("received signal %d\n", signo);
    }
}
```

```c
int main(void) {
    if (signal(SIGINT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGQUIT, sig_usr) == SIG_ERR) {
        printf("can't catch SIGINT\n");
        exit(-1);
    }
    if (signal(SIGUSR1, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR1\n");
        exit(-1);
    }
    if (signal(SIGUSR2, sig_usr) == SIG_ERR) {
        printf("can't catch SIGUSR2\n");
        exit(-1);
    }
    if (signal(SIGTSTP, sig_usr) == SIG_ERR) {
        printf("can't catch SIGTSTP\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}
```

compile & run

```
(base) mahmutunan@MacBook-Pro ~ % cd Desktop/lecture23
(base) mahmutunan@MacBook-Pro lecture23 % ls
sighandler      sighandler.c    sigusr          sigusr.c
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sighandler.c -o sighandler
(base) mahmutunan@MacBook-Pro lecture23 % ./sighandler
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3
^Zreceived SIGTSTP signal 18
^Creceived SIGINT signal 2
^Creceived SIGINT signal 2
^Zreceived SIGTSTP signal 18
^\received SIGQUIT signal 3
```

infinite loop

- Notice that we have replaced the default signal handlers to kill this job from the keyboard and the program is in an infinite loop with pause. As a result we cannot terminate this program from the keyboard. We have to login to the specific machine this process is running and kill this process using either *kill* or *top* commands. You can follow the examples from the first part and kill this process.

Default Action Of Signals

- Each signal has a default action, one of the following:
  - **Term:** The process will terminate.
    Core: The process will terminate and produce a core dump file.
    Ign: The process will ignore the signal.
    Stop: The process will stop.
    Cont: The process will continue from being stopped.
- Default action may be changed using handler function. Some signal's default action cannot be changed. **SIGKILL** and **SIGABRT** signal's default action cannot be changed or ignored
- https://linuxhint.com/signal_handlers_c_programming_language/

Basic Signal handler examples

```c
#include<stdio.h>
#include<signal.h>
#include<unistd.h>
void sig_handler(int signum){

  //Return type of the handler function should be void
  printf("\nInside handler function\n");
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example1.c -o Example1
tump@tump:~/Desktop/c_prog/signal$ ./Example1
1 : Inside main function
2 : Inside main function
^C
Inside handler function
3 : Inside main function
4 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

```c
#include<stdio.h>
#include<unistd.h>
#include<signal.h>
int main(){
  signal(SIGINT,SIG_IGN); // Register signal handler for ignoring the signal

  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example2.c -o Example2
tump@tump:~/Desktop/c_prog/signal$ ./Example2
1 : Inside main function
2 : Inside main function
^C3 : Inside main function
4 : Inside main function
^C5 : Inside main function
^\Quit (core dumped)
tump@tump:~/Desktop/c_prog/signal$
```

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

void sig_handler(int signum){
  printf("\nInside handler function\n");
  signal(SIGINT,SIG_DFL);   // Re Register signal handler for default action
}

int main(){
  signal(SIGINT,sig_handler); // Register signal handler
  for(int i=1;;i++){    //Infinite loop
    printf("%d : Inside main function\n",i);
    sleep(1);  // Delay for 1 second
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example3.c -o Example3
tump@tump:~/Desktop/c_prog/signal$ ./Example3
1 : Inside main function
2 : Inside main function
3 : Inside main function
^C
Inside handler function
4 : Inside main function
^C
tump@tump:~/Desktop/c_prog/signal$
```

```c
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);
  kill(getppid(),SIGUSR1);
}

int main(){
  pid_t pid;
  if((pid=fork())<0){
    printf("Fork Failed\n");
    exit(1);
  }
  /* Child Process */
  else if(pid==0){
    signal(SIGUSR1,sig_handler_child); // Register signal handler
    printf("Child: waiting for signal\n");
    pause();
  }
  /* Parent Process */
  else{
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
  }
  return 0;
}
```

```c
#include<stdio.h>
#include <unistd.h>
#include <stdlib.h>
#include<signal.h>
void sig_handler_parent(int signum){
  printf("Parent : Received a response signal from child \n");
}

void sig_handler_child(int signum){
  printf("Child : Received a signal from parent \n");
  sleep(1);
  kill(getppid(),SIGUSR1);
}

int main(){
  pi
  if
  }
  /*
  el
  }
  /*
  el
    signal(SIGUSR1,sig_handler_parent); // Register signal handler
    sleep(1);
    printf("Parent: sending signal to Child\n");
    kill(pid,SIGUSR1);
    printf("Parent: waiting for response\n");
    pause();
  }
  return 0;
}
```

```
tump@tump:~/Desktop/c_prog/signal$ gcc Example6.c -o Example6
tump@tump:~/Desktop/c_prog/signal$ ./Example6
Child: waiting for signal
Parent: sending signal to Child
Parent: waiting for response
Child : Received a signal from parent
Parent : Received a response signal from child
tump@tump:~/Desktop/c_prog/signal$
```

Exercise Sigint

- Till now we used the kill command and the keyboard to generate the signals.
- Now we will generate the signal in a program using the C API for *kill* or *raise* system call.
- In this example, we replace the SIGINT signal handler with our own and in the signal handler asks the user if the process should be terminated and then based on the response continue with either terminating the process or let it continue to run.
- We use read function instead of scanf to emphasize that scanf is not a reentrant function

sigint

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

static void sig_int(int signo) {
    ssize_t n;
    char buf[2];

    signal(signo, SIG_IGN); /* ignore signal first */
    printf("Do you really want to do this: [Y/N]? ");
    fflush(stdout);
    n = read(STDIN_FILENO, buf, 2);
    if ( buf[0] == 'Y' ) {
        raise(SIGTERM); // or kill(0, SIGTERM); // or exit (-1);
    } else {
        printf("Ignoring signal, be careful next time!\n");
        fflush(stdout);
    }
    signal(signo, sig_int); /* reinstall the signal handler */
}
```

```c
int main(void) {
    if (signal(SIGINT, sig_int) == SIG_ERR) {
        printf("Unable to catch SIGINT\n");
        exit(-1);
    }
    for ( ; ; )
        pause();

    return 0;
}

```

```
(base) mahmutunan@MacBook-Pro lecture23 % gcc -Wall sigint.c -o sigint
(base) mahmutunan@MacBook-Pro lecture23 % ./sigint
^CDo you really want to do this: [Y/N]? N
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]?
Ignoring signal, be careful next time!
^CDo you really want to do this: [Y/N]? Y
zsh: terminated  ./sigint
(base) mahmutunan@MacBook-Pro lecture23 %
```

recall -forkexecvp.c

```c
int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
        printf("If you see this statement then execl failed ;-(\n");
    perror("execvp");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
        printf("Wait for the child process to terminate\n");
        wait(&status); /* wait for the child process to terminate */
        if (WIFEXITED(status)) { /* child process terminated normally */
            printf("Child process exited with status = %d\n", WEXITSTATUS(status));
        } else { /* child process did not terminate normally */
            printf("Child process did not terminate normally!\n");
            /* look at the man page for wait (man 2 wait) to determine
               how the child process was terminated */
        }
    } else { /* we have an error */
        perror("fork"); /* use perror to print the system error message */
        exit(EXIT_FAILURE);
    }

    printf("[%ld]: Exiting program .....\n", (long)getpid());

    return 0;
}
```

- Let us now consider how signals impact child processes created using fork/exec. We will be running a c program that consist of fork and execvp system calls (hw1.c)
- This code illustrates the use of dynamic memory allocation to create contiguous 2D-matrices and use traditional array indexing. It also illustrate the use of gettime to measure wall clock time.

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^C
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.3  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19320  0.0  0.0 161588  1868 pts/0    R+   10:06   0:00 ps -u
$
```

```
$ ./a.out /home/UAB/puri/CS332/shared/hw1 1000
Wait for the child process to terminate
^Z
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.1  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri     19352 29.0  0.6  27800 23844 pts/0    T    10:08   0:01 /home/UAB/puri/
puri     19353  0.0  0.0 161588  1864 pts/0    R+   10:08   0:00 ps -u
$ kill -CONT 19352
$
$ ps -u
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
puri     19202  0.0  0.1 125440  3912 pts/0    Ss   10:05   0:00 -bash
puri     19351  0.0  0.0   4220   352 pts/0    T    10:08   0:00 ./a.out /home/U
puri     19352 17.9  0.6  27800 23844 pts/0    R    10:08   0:04 /home/UAB/puri/
puri     19355  0.0  0.0 161588  1872 pts/0    R+   10:08   0:00 ps -u
$ Time taken for size 1000 = 32.274239 seconds
$ jobs
[1]+  Stopped                 ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ kill -CONT 19351
$ Child process exited with status = 0
[19351]: Exiting program .....
[1]+  Done                    ./a.out /home/UAB/puri/CS332/shared/hw1 1000
$ jobs
$
```

Linux I/O Streams

- Before we discuss I/O redirection, let us review how I/O is handled in Linux systems.
- Each process in a Linux environment has three different file descriptors available when a process is created: standard input (*stdin – 0*), standard output (*stdout – 1*), and standard error (*stderr – 2*).
- These three file descriptors are created when a process is created.
- We use the *stdin* file descriptor to read input from a keyboard or from another a file or from another program.
- Similarly, we use the *stdout* and *stderr* file descriptors to write output and error messages, respectively, to the terminal.

- Input and output in the Linux environment is distributed across three streams. These streams are:
- **standard input** (**stdin**)
- **standard output** (**stdout**)
- **standard error** (**stderr**)

- The streams are also numbered:
- **stdin** (**0**)
- **stdout** (**1**)
- **stderr** (**2**)

During standard interactions between the user and the terminal, standard input is transmitted through the user's keyboard. Standard output and standard error are displayed on the user's terminal as text. Collectively, the three streams are referred to as the *standard streams*.

https://www.digitalocean.com/community/tutorials/an-introduction-to-linux-i-o-redirection

cat command

```
(base) mahmutunan@MacBook-Pro lecture26 % cat
1
1
2
2
3
3
(base) mahmutunan@MacBook-Pro lecture26 %
```

Stream Redirection

- **Overwrite**

  > - standard output

  < - standard input

  2> - standard error

- **Append**

  >> - standard output

  << - standard input

  2>> - standard error

```
(base) mahmutunan@MacBook-Pro lecture26 % cat > outputFile.txt
1
2
3
4
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat >> outputFile.txt
a
b
c
(base) mahmutunan@MacBook-Pro lecture26 % cat outputFile.txt
1
2
3
4
a
b
c
```

```
(base) mahmutunan@MacBook-Pro lecture26 % ls >> listOffiles.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat listOffiles.txt
error.txt
forkexecvp
forkexecvp.c
forkexecvp2.c
hw2.c
input.txt
ioredirect.c
lab7_solution.c
listOffiles.txt
myprog
myprog.c
output.txt
output2.txt
outputFile.txt
```

stdout stderr

```
(base) mahmutunan@MacBook-Pro lecture26 % echo "some output using stdout"
some output using stdout
(base) mahmutunan@MacBook-Pro lecture26 % echo

(base) mahmutunan@MacBook-Pro lecture26 % _
```

```
(base) mahmutunan@MacBook-Pro lecture26 % cat nonexistingfile.txt
cat: nonexistingfile.txt: No such file or directory
(base) mahmutunan@MacBook-Pro lecture26 % _
```

- You have already been using these file descriptors when you wrote the insertion sort program in C.
- You read the number of elements and the elements to be sorted from the keyboard using the *scanf* function.
- The *scanf* function was using *stdin* file descriptor to read your keyboard input. In other words, the following two functions are equivalent:

```
scanf("%d", &N);
fscanf(stdin, "%d", &N);
```

- Similarly when you use the *printf* function to print the output of your program you are using the *stdout* file descriptor.

- The two functions below are equivalent:

```
printf("%d\n", N);
fprintf(stdout,"%d\n", N);
```

- The file descriptors *stdin*, *stdout*, and *stderr* are defined in the header file *stdio.h*. We typically use the *stderr* file descriptor to write error messages.

- If we need to save the output or error message from a program to a file or read data from a file instead of entering it through the keyboard, we can use the I/O redirection supported by the Linux shell (such as bash).
- In fact, we already used this in one of the earlier labs when we used insertion sort to sort large input values.
- The following examples show how to use I/O redirection in the bash shell to read input from a file (instead of entering it from the keyboard) and send the output and error messages to different files (instead of the terminal)

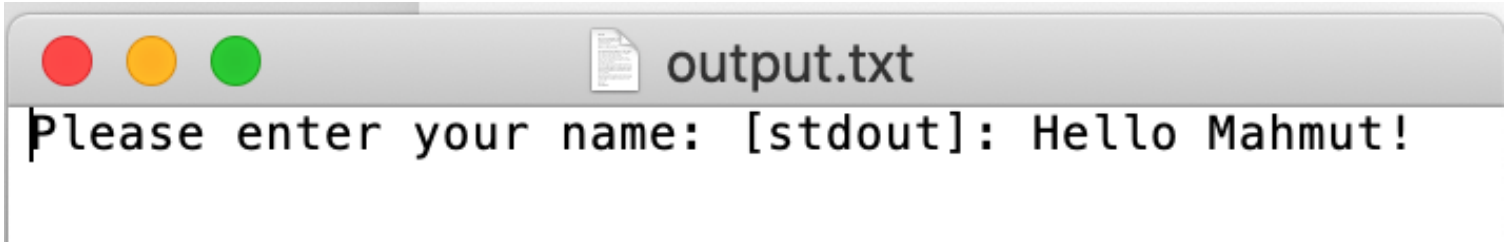
Exercise 1

```c
#include <stdio.h>

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

    printf("Please enter your name: ");
    scanf("%s", name);
    printf("[stdout]: Hello %s!\n", name);
    fprintf(stderr, "[stderr]: Hello %s!\n", name);

    return 0;
}
```

Compile the program myprog.c,
create a file called *input.txt*, type you name in the file *input.txt*

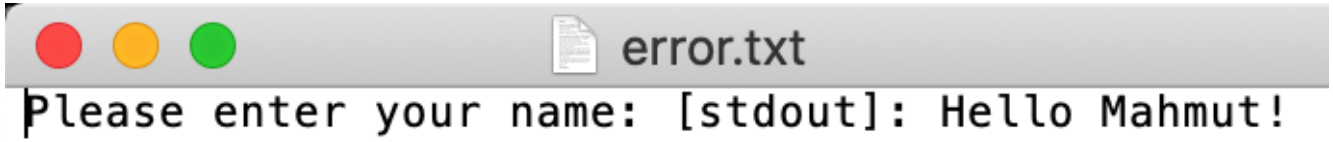
compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % touch input.txt
(base) mahmutunan@MacBook-Pro lecture26 % echo "Mahmut" > input.txt
(base) mahmutunan@MacBook-Pro lecture26 % cat input.txt
Mahmut
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall myprog.c -o myprog
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt > output.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % _
```



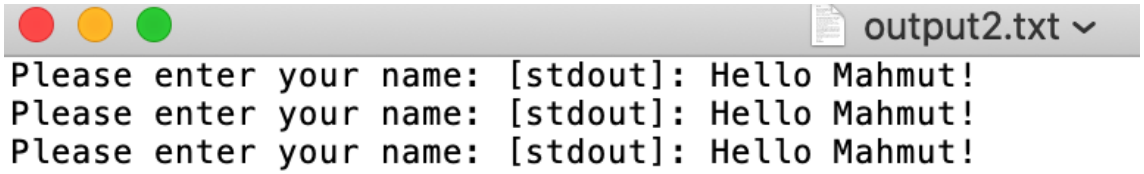
compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt
Please enter your name: [stdout]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt 2> error.txt >output.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >output2.txt 2> error.txt
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt & >output2.txt 2> error.txt
[1] 91593
Please enter your name: [stdout]: Hello Mahmut!
[stderr]: Hello Mahmut!
[1]  + done       ./myprog < input.txt
^C
(base) mahmutunan@MacBook-Pro lecture26 %
```



compile & run

```
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 % ./myprog < input.txt >> output2.txt
[stderr]: Hello Mahmut!
(base) mahmutunan@MacBook-Pro lecture26 %
```



You can replace **>** with **>>** if you like to append to the file instead of overwriting the file

Sharing between parent and child processes

- When we created a new process using fork/exec in the previous labs, we noted that the child process is a copy of the parent process and it inherits several attributes from the parent process such as open file descriptors.
- This duplication of descriptors allowed the child processes to read and write to the standard I/O streams (note that the child process was able to read input from the keyboard and write output to the terminal).
- As a result of this sharing both parent and child processes share the three standard I/O streams: *stdin*, *stdout*, and *stderr*.

Exercise 2

- Let us use the example from the previous lectures to illustrate this by adding the following lines in the parent process:

```
    char name[BUFSIZ];

        printf("Please enter your name: ");
        scanf("%s", name);
        printf("[stdout]: Hello %s!\n", name);
        fprintf(stderr, "[stderr]: Hello
%s!\n", name);
```

Exercise 2

```c

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/wait.h>

int main(int argc, char **argv) {
    pid_t pid;
    int status;

    if (argc < 2) {
        printf("Usage: %s <command> [args]\n", argv[0]);
        exit(-1);
    }

    pid = fork();
    if (pid == 0) { /* this is child process */
        execvp(argv[1], &argv[1]);
    perror("exec");
    exit(-1);
    } else if (pid > 0) { /* this is the parent process */
```

Exercise 2

```
23          char name[BUFSIZ];
24
25          printf("[%d]: Please enter your name: ", getpid());
26          scanf("%s", name);
27          printf("[stdout]: Hello %s!\n",  name);
28          fprintf(stderr, "[stderr]: Hello %s!\n",  name);
29
30          wait(&status); /* wait for the child process to terminate */
31          if (WIFEXITED(status)) { /* child process terminated normally */
32              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33          } else { /* child process did not terminate normally */
34              printf("Child process did not terminate normally!\n");
35              /* look at the man page for wait (man 2 wait) to determine
36                 how the child process was terminated */
37          }
38      } else { /* we have an error */
39          perror("fork"); /* use perror to print the system error message */
40          exit(EXIT_FAILURE);
41      }
42
43      return 0;
44  }
45
```

compile & run

- If we compile and execute the program by using *myprog* (used earlier) as the child process, we will notice that the prompt to enter the name is printed twice. If we enter the name, which process is reading the keyboard input?

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[86530]: Please enter your name: Please enter your name: mahmut
[stdout]: Hello mahmut!
[stderr]: Hello mahmut!
```

Exercise 2

```
23          char name[BUFSIZ];
24
25          printf("[%d]: Please enter your name: ", getpid());
26          scanf("%s", name);
27          printf("[%d-stdout]: Hello %s!\n", getpid(), name);
28          fprintf(stderr, "[%d-stderr]: Hello %s!\n", getpid(), name);
29
30          wait(&status); /* wait for the child process to terminate */
31          if (WIFEXITED(status)) { /* child process terminated normally */
32              printf("Child process exited with status = %d\n", WEXITSTATUS(status));
33          } else { /* child process did not terminate normally */
34              printf("Child process did not terminate normally!\n");
35              /* look at the man page for wait (man 2 wait) to determine
36                 how the child process was terminated */
37          }
38      } else { /* we have an error */
39          perror("fork"); /* use perror to print the system error message */
40          exit(EXIT_FAILURE);
41      }
42
43      return 0;
44  }
```

compile & run

- We could add the PID in the printf statement to make this explicit. We can update the code above to print the PID and test the program. In any case, this illustrates the result of sharing of the standard I/O streams between the parent and the child processes

```
(base) mahmutunan@MacBook-Pro lecture26 % gcc -Wall forkexecvp2.c -o forkexecvp
(base) mahmutunan@MacBook-Pro lecture26 % ./forkexecvp ./myprog
[85953]: Please enter your name: Please enter your name: mahmut
[85953-stdout]: Hello mahmut!
[85953-stderr]: Hello mahmut!
```

- If we like to change the behavior of all child processes to use separate files instead of the standard I/O streams we have to replace the standard I/O file descriptors with new file descriptors.
- We will use the dup2() system call to create a copy of this file descriptors and associate separate files to replace the standard I/O streams.

dup2()

- **Copying file descriptors – dup2() system call**
- The dup2() system call duplicates an existing file descriptor and returns the duplicate file descriptor.
- After the call returns successfully, both file descriptors can be used interchangeably. If there is an error then -1 is returned and the corresponding errno is set (look at the man page for dup2() for more details on the specific error codes returned).
- The prototype for the dup2() system call is shown below:

```
#include <unistd.h>

int dup2(int oldfd, int newfd);
```

dup() vs dup2()

- These system calls create a copy of the file descriptor *oldfd*.
- **dup**() uses the lowest-numbered unused descriptor for the new descriptor.
- **dup2**() makes *newfd* be the copy of *oldfd*, closing *newfd* first if necessary, but note the following:
  - If *oldfd* is not a valid file descriptor, then the call fails, and *newfd* is not closed.
  - If *oldfd* is a valid file descriptor, and *newfd* has the same value as *oldfd*, then **dup2**() does nothing, and returns *newfd*.
- After a successful return from one of these system calls, the old and new file descriptors may be used interchangeably. T