System Programming

Clinic Session

Essential Linux Commands

- ls – list all contents
- cd – change directory
- mkdir – create new directory
- touch – create new file
- rm – delete file
- rmdir – delete empty directory

Code Execution

1. gcc source.c –o executable_file_name
2. ./executable_file_name

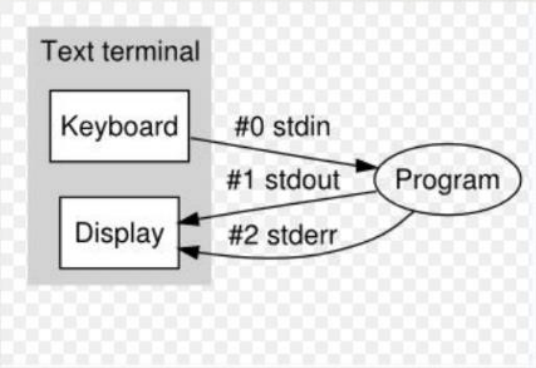
File Access

Low Level File Access – System Call

- **System Default File Descriptors**

When a program starts, there are three file descriptors* automatically opened:

0	standard input
1	standard output
2	standard error



*A file descriptor is an abstract key or a reference for accessing a file.

Low Level File Access – System Call

- **Open**
  - int open(const char *pathName, int oflags, mode_t mode);
    - Success : return fileDescriptor
    - Fail : return -1

oflags

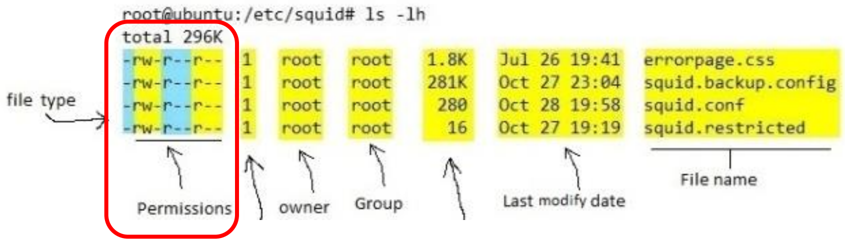
Mandatory access modes:

O_RDONLY	Open for read-only
O_WRONLY	Open for write-only
O_RDWR	Open for reading and writing

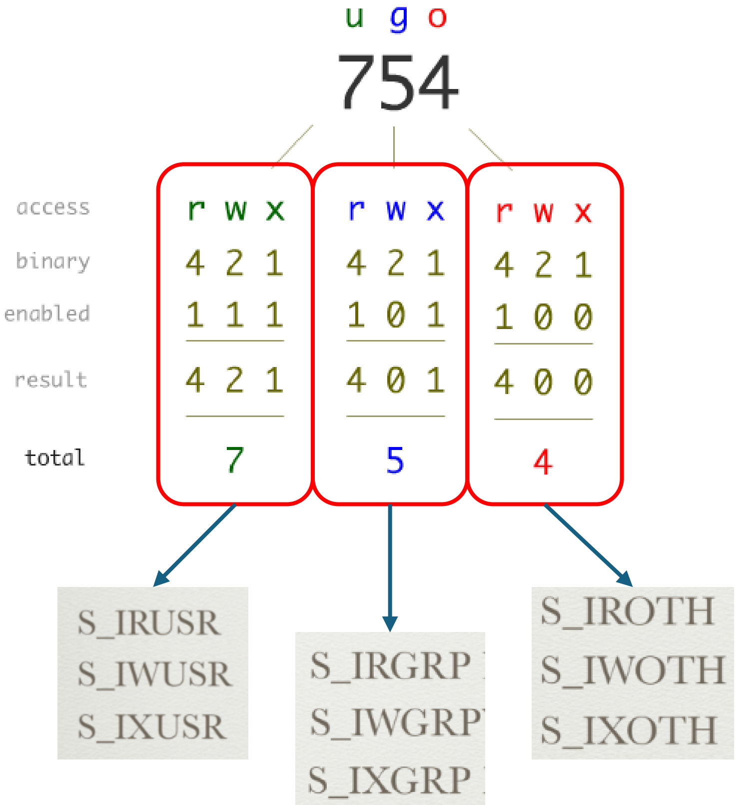
O_APPEND	Place written data at the end of the file.
O_TRUNC	Set the length of the file to zero, discarding existing contents.
O_CREAT	Create the file, if necessary, with permissions given in mode.
O_EXCL	Used with O_CREAT, ensures that the caller creates the file. The open is atomic, i.e. it's performed with just one function call. This protects against two programs creating the file at the same time. If the file already exists, open will fail.

mode

S_IRUSR	Read permission for owner	S_IRGRP	Read permission for group
S_IWUSR	Write permission for owner	S_IWGRP	Write permission for group
S_IXUSR	Execute permission for owner	S_IXGRP	Execute permission for group
S_IROTH	Read permission for others		
S_IWOTH	Write permission for others		
S_IXOTH	Execute permission for others		



Low Level File Access – System Call



Ex. int out = open("file.out", O_WRONLY | O_CREAT | O_TRUNC, 0666);

Low Level File Access – System Call

- **Write**
  - size_t write(int fileDescriptor, const void* buffer, size_t byteCount);

    Can be {0 : stdin, 1 : stdout, 2 : stderr}
    - <u>Success</u> : return number of bytes written
    - <u>Fail</u> : return -1
- **Read**
  - size_t read(int fileDescriptor, const void* buffer, size_t byteCount);

    Can be {0 : stdin}
    - <u>Success</u> : return number of bytes read
    - <u>Fail</u> : return -1

Low Level File Access – System Call

- **Close**
  - int close(int fileDescriptor);
    - Success : return 0
    - Fail : return -1

Low Level File Access – System Call

- **Lseek**
  - off_t lseek(int fileDescriptor, off_t byteLength, int referencePosition);
    - Success : return offset location as measured in bytes from the beginning of the file
    - Fail : return -1

referencePosition (whence)

SEEK_SET:	the beginning of the file
SEEK_CUR:	the current position
SEEK_END:	the end of the file

High Level File Access – Function Call

- **Library Default File Pointers**

When a program starts, there are three streams* automatically opened with file pointers** pointed to each stream:

stdin	->	standard input
stdout	->	standard output
stderr	->	standard error

*A stream is a representation of an associated file.

**A file pointer is an abstract key or a reference for accessing a file, which is equivalent to the low-level file descriptor.

High Level File Access – Function Call

- **Buffering**
  - Fully Buffered : I/O is performed when buffer is filled
  - Line Buffered : I/O is performed after new line character is encountered
  - Unbuffered : I/O is performed without buffering
- **Change buffering type**
  - void setbuf(FILE* filePointer, char* buffer);
  - int setvbuf(FILE* filePointer, char* buffer, int mode, size_t size);
    - <u>Success</u> : return 0
    - <u>Fail</u> : return non-zero

High Level File Access – Function Call

Function	Mode	Buf	Buffer & Length	Type of buffering
setbuf		Non-Null	User buf of length BUFSIZ (defined in stdio.h)	Fully buffered or line buffered
		NULL	(no buffer)	unbuffered
setvbuf	_IOFBF	Non-Null	User *buf* of length *size*	Fully buffered
		NULL	System buffer of appropriate length	
	_IOLBF	Non-Null	User *buf* of length *size*	Line buffered
		NULL	System buffer of appropriate length	
	_IONBF	(ignored)	(no buffer)	unbuffered

High Level File Access – Function Call

- **fopen**
  - FILE* fopen(const char *pathName, const char *type);
    - Success : return filePointer(stream)

Type	Description
r or rb	Open for reading
w or wb	Truncate to 0 length or create for writing
a or ab	Append; open for writing at the end of file, or create for writing
r+ or r+b or rb+	Open for reading and writing
w+ or w+b or wb+	Truncate to 0 length or create for reading and writing
a+ or a+b or ab+	Open or create for reading and writing at the end of file

High Level File Access – Function Call

- **fwrite**
  - size_t fwrite(const void* buffer, size_t bufferSize, size_t count, FILE* stream);
    - Success : return number of successfully written items
    - Fail or EOF : return short item count (or zero).
- **fread**
  - size_t fread(void *buffer, size_t bufferSize, size_t byteCount, FILE *stream);
    - Success : return number of successfully written items
    - Fail or EOF : return short item count (or zero).

High Level File Access – Function Call

- **fclose**
  - int fclose(FILE *stream)
    - Success : return 0
    - Fail : return EOF

High Level File Access – Function Call

- **fseek**
  - int fseek(FILE *stream, long offset, int referencePosition);
    - Success : return 0
    - Fail : return -1

referencePosition (whence)

SEEK_SET: the beginning of the file
SEEK_CUR: the current position
SEEK_END: the end of the file

High Level File Access – Function Call

- **int fflush(FILE *stream);**
  - clear output buffer and move buffered data to console or disk
- **int fgetc(FILE* stream);**
  - reads next character from stream
- **int getc(FILE* stream);**
  - Similar to getc();
- **int getchar(void);**
  - getc(stdin);
- **int ungetc(int c, FILE* stream);**
  - pushes character back to stream cast to unsigned char, the characters will be returned in reverse order

High Level File Access – Function Call

- **int fputc(int c, FILE *stream);**
  - writes the character cast to an unsigned character to stream
- **int putc(int c, FILE *stream);**
  - Similar to fputc();
- **int putchar(int c);**
  - putc(c, stdout);

System Call vs Function Call

- System call is implemented in Linux kernel, and will takes over execution of program until call completes
- Function call is functions in library, the arguments placed in registers or in stack and execution transferred to the start of function’s code

Multiple Processes

Program Address Space

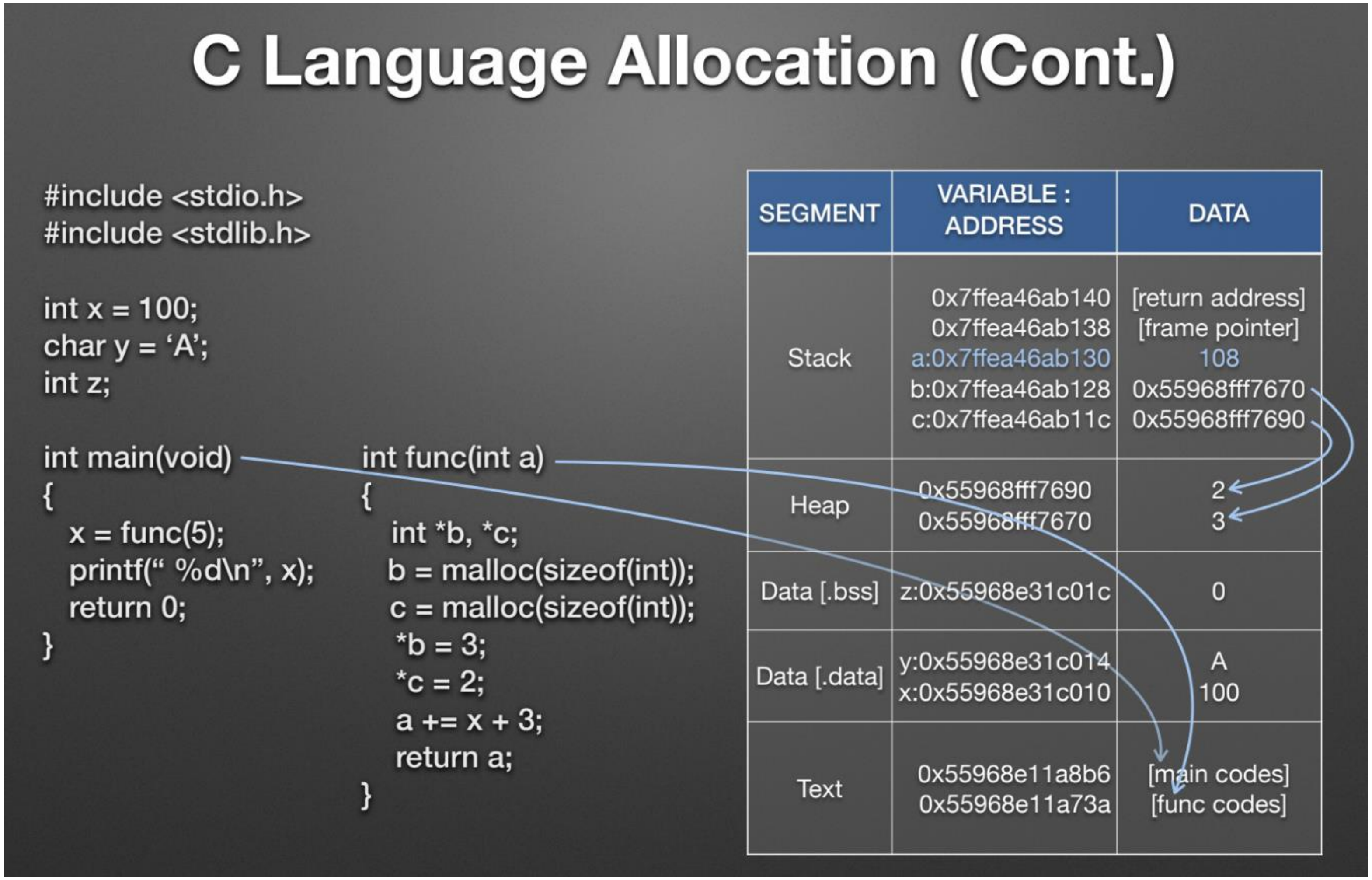
- **Text Segment**
  - Code / function as text
- **Data Segment**
  - .data
    - Initialized global and static data
    - int x = 100;
    - char name[] = “bob”;
  - .bss (block started by symbol)
    - Uninitialized global and static data
    - int array[100];
    - int z;

Program Address Space

- **Heap**
  - managed by malloc(), realloc(), and free()
- **Stack**
  - Parameters and return address of the function and local variables

** pointer variable initialized by malloc() in main() or function() will be stored in STACK ➔ local variable
** the pointed value will be stored in HEAP ➔ static variable

Program Address Space



C Language Allocation (Cont.)

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

int x = 100;
char y = 'A';
int z;

int main(void)
{
   x = func(5);
   printf(" %d\n", x);
   return 0;
}

int func(int a)
{
   int *b, *c;
   b = malloc(sizeof(int));
   c = malloc(sizeof(int));
   *b = 3;
   *c = 2;
   a += x + 3;
   return a;
}
```

SEGMENT	VARIABLE : ADDRESS	DATA
Stack	0x7ffea46ab140 0x7ffea46ab138 a:0x7ffea46ab130 b:0x7ffea46ab128 c:0x7ffea46ab11c	[return address] [frame pointer] 108 0x55968fff7670 0x55968fff7690
Heap	0x55968fff7690 0x55968fff7670	2 3
Data [.bss]	z:0x55968e31c01c	0
Data [.data]	y:0x55968e31c014 x:0x55968e31c010	A 100
Text	0x55968e11a8b6 0x55968e11a73a	[main codes] [func codes]

fork

- **pid_t fork();**
  - Create new child process
  - Child process gets the copy of these from the parent process
    - Data segment
    - Heap
    - Stack
    - File descriptor
  - Return value
    - 0 : this is the new child process
    - > 1 : this is parent process
    - -1 : failed

vfork

- **pid_t vfork()**
  - Create new child process
  - Child process gets the copy of these from the parent process
    - Data segment
    - Heap
    - Stack
    - File descriptor
  - Return value
    - 0 : this is the new child process
    - > 1 : this is parent process
    - -1 : failed

Wait and waitpid

- To find out the exit status of child processes
- **wait(int * status);**
  - blocks until it receives the exit status from a child
  - **status:**
    - WIFEXITED true when process exited normally
    - WIFSIGNALED true when process killed by a signal
- **waitpid(pid_t pid, int * status, int options);**
  - **pid :**
    - < -1 wait for any child whose gpid is the same as |pid|
    - -1 wait for any child to terminate
    - 0 wait for a child in the same process group as the current process
    - > 0 wait for specified process pid to exit

Zombie and Orphan

- **Zombie**
  - child end but parent do not call wait(); or waitpid();
  - defunct
- **Orphan**
  - parent process end but child not
  - Child will be adopted

exec

- replace the current program running within the process with another program
- **int execl(const char *path, const chat * argv0, argv1,...., NULL);**
    - execlp
    - execle
- **int execv(const char* path, char *const argv[]);**
    - execvp
    - execve

char* argv[] = {“print”, “1”, “10”, NULL};

Multiple Threads

Process and Thread

Processes	Threads
Copy address space	Share visual space in address space
Can not harm other process	Errant thread can harm other threads since they share same resources
Can use exec(); to run different executable	Must run the same executable
	For fine-grained parallelism (for nearly identical tasks)
Need IPC mechanisms to share data	Sharing data is trivial

pthread_create

- **int pthread_create(pthread_t * threadID, const pthread_att_t * attribute, void *function, void * parameter);**
  - create new thread
  - <u>Success</u>: return 0
  - <u>Fail</u>: return error number

Thread Attribute : detached state

- **Detached state attribute**
    - Joinable(default) Thread’s exit state hangs around in the system until another thread calls pthread_join to obtain its return value.
    - detached thread is <u>cleaned up automatically</u> when it terminates.

```
int main ()
{
   pthread_attr_t attr;
   pthread_t thread;
   pthread_attr_init(&attr);
   pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
   pthread_create(&thread, &attr, &thread_function, NULL);
   pthread_attr_destroy(&attr);

   /* Do work here... */

   /* No need to join the second thread.  */
   /* However the main thread needs to make sure that it will finish after the other threads */
   return 0;
}
```

MultiThread : Parsing Arguments

- **int pthread_create(pthread_t * threadID, const pthread_att_t * attribute, void *function, void * parameter);**
    - Single parameter is ok
    - Parse multiple parameters as a <u>struct</u>
        - Type casting****

```c
struct char_print_params
{
    char ch;
    int count;
};
```

```c
void* char_print(void* parameters)
{
    struct char_print_params* p = (struct char_print_params*) parameters;
    int i;
    for(i = 0; i < p->count; i++){
        fputc(p->ch, stderr);
    }
    return NULL;
}
```

```c
int main(){
    pthread_t threadID1, threadID2;
    struct char_print_params threadArgs1;
    struct char_print_params threadArgs2;

    threadArgs1.ch = 'x';
    threadArgs1.count = 30000;
    pthread_create(&threadID1, NULL, &char_print, &threadArgs1);

    threadArgs2.ch = 'o';
    threadArgs2.count = 20000;
    pthread_create(&threadID2, NULL, &char_print, &threadArgs2);

    return 0;
}
```

MultiThread : Join / Return Value

- **pthread_join(pthread_t* threadID, void** returnValue);**
    - Works similar to wait()
    - Get the return value from thread function

MultiThread : Thread Cancellation

- When multiple threads are running in a process, thread cancellation permits one thread to cancel another thread in that process.
  - **Asynchronously cancelable:** can be canceled at any point in its execution
  - **Synchronously cancelable (default):** cancellation requests are queued, and the thread is canceled only when it reaches specific points in its execution.
  - **Uncancelable:** cancellation requests are quietly ignored

MultiThread : Thread Cancellation

- When pthread_cancel() is called

Thread Cancellation Type	Argument of pthread_setcanceltype	Need the cancellation point
Asynchronously cancelable	PTHREAD_CANCEL_ASYNCHRONOUS	No
Synchronously cancelable	PTHREAD_CANCEL_DEFERRED	Yes with pthread_testcancel function
Uncancelable	PTHREAD_CANCEL_DISABLE	No

MultiThread : Thread-specific Data

- retrieves thread-specific data, which is a unique value stored for each thread. In simpler terms, it allows each thread to access its own version of a variable, even if multiple threads are running the same code.

Thread Specific-Data Functions	**Objectives**
static pthread_key_t thread_log_key	Define Global Variable Key
pthread_key_create (&thread_log_key, close_thread_log)**	Create the magic key
pthread_setspecific (thread_log_key, thread_log);	Associate or set the magic key to its own data (thread_log)
FILE* thread_log = (FILE*) pthread_getspecific (thread_log_key);	Use the magic key to get its own data (thread_log)

Race Conditions

Race Conditions

- A race condition occurs when multiple processes are trying to do something with shared resource and the final outcome depends on the order in which the processes run.

Signals

Signals: Sending Signals Using Terminal

- **ps –u**
  - Get the running process’ PID
- **kill –l**
  - List all signal_names and associated signal_no
- **kill -<signal_name/signal_no> <PID>**
  - send specific signal to PID

Signals: Common Signals

- Common signals :
  - SIGHUB : hangup
  - SIGINT : interrupt
  - SIGFPE : arithmetic error / floating point error
  - SIGKILL : terminate process immediately
  - SIGTERM : terminate process allows the process to perform any necessary cleanup
  - SIGCHLD : child process terminated

Signals: Common Signals

- 4 things to do when signals occur
  - Ignore the signal (SIGKILL and SIGSTOP cannot be ignored)
  - Response according to the handler routine set by user
  - Block the signals
  - Let the default action (usually happens in process termination)

Signals: Signal Handling

- pair signal and signal handler
- **void (*signal(int signo, void (*func) (int))) (int);**
    - ***func** can be
        - SIG_DFL : default handling function depends on OS
        - SIG_IGN : ignore signal
        - User defined functions
    - return default value of signal handler function, if error return SIG_ERR

Signals: Signal Handling

- suspending the calling process until a signal is received
    - **int pause(void);**
- Sending signals
    - **int raise(int sig);** **// send signal to itself**
    - **int kill(pid_t pid, int sig);** **// send signal to specific process**

getpid(); // get current process id
getppid(); // get parent process id

- Printing out signal information
    - **void psignal(int sig, const char *s);**

Signals: Signal Handling

- suspending the calling process until a signal is received
  - **int pause(void);**
- Sending signals
  - **int raise(int sig);** **// send signal to itself**
  - **int kill(pid_t pid, int sig);** **// send signal to specific process**

getpid(); // get current process id
getppid(); // get parent process id

- Printing out signal information
  - **void psignal(int sig, const char *s);**

POSIX Signals

- A signal set is a bitmask of signals that you want to block
- Each bit in the bitmask corresponds to a given signal in kill –l (bit 10 == SIGUSR1)
- In POSIX signals, a blocked signal is not thrown away, but treated as pending for to the process to unmask that bit to accept that signal.

POSIX Signals

- **struct sigaction**: POSIX signal struct
- **int sigaction (int signal_no, const struct sigaction *new_act, struct sigaction *old_act);**
    - Pair signal with the new signal struct

```
strcut{
        void (*sa_hander)(int);
        void (*sa_sigaction)(int, siginfo_t*, void*);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restoreer)(void);
};
```

POSIX Signals

- **struct sigaction**: POSIX signal struct

```
strcut{
        void (*sa_hander)(int);
        void (*sa_sigaction)(int, siginfo_t*, void*);
        sigset_t sa_mask;
        int sa_flags;
        void (*sa_restoreer)(void);
};
```

  - **sa_handler**: set the signal handler
  - **sa_mask**: used to set the bits of the signals to be block
  - **sa_flags**:
    - sa_flags
      - * SA_RESTART flag to automatically restart interrupted system calls
      - * SA_NOCLDSTOP flag to turn off SIGCHLD signaling when children die.
      - * SA_RESETHAND clears the handler (ie. resets the default) when the signal is delivered (recidivist).
      - * SA_NOCLDWAIT flag on SIGCHLD to inhibit zombies.
      - * SA_SIGINFO flag indicates use value in sa_sigaction over sa_handler

POSIX Signals

- **int sigemptyset(sigset_t * set);**
  - Clear all bits to 0
- **int sigfillset(sigset_t * set);**
  - Set all bits to 1
- **int sigaddset(sigset_t * set, int signal_no);**
  - Adds a particular signal_no to the set
- **int sigdelset(sigset_t * set, int signal_no);**
  - Unmasks signal_no from the set

Reentrant

- Reentrant function is one that can be interrupted and re-entered by another thread without any ill-effects (i.e. values inconsistency).
- Must hold no static data, must not return a pointer to static data, must work only on the data provided to it by the caller, and must not call other non-reentrant functions.

Exam Guidelines

- File Access
    - Code analysis => lseek : purposes / how it works / output
- Multiple Processes
    - Definition
    - Program’s Address Space ***
    - Zombie / Orphan => definition / code
    - fork / exec => parent and child do different things
- Multiple Threads
    - Definition => different between thread and process (concurrency and parallelism) / … / … / ….
    - Parsing data***
- Signals??????????
    - Code analysis????????????

Coding Practice Sample

- Multiple Processes
  - Using both fork and exec, 1. parent exec the print function to print “Parent” 123 times / 2. child exec print function to print “Child” 321 times. **both processes should share the same print function.
  - write the code that generates zombie/orphan processes
- Multiple Threads
  - Write the multi-thread code that 1. thread 1 prints ‘a’ 999 times / 2. thread 2 prints ‘b’ 888 times / 3. thread 3 prints ‘c’ 777 times. **all threads should share the same thread function.