Creating process

A new process can be created using fork system call. The new process is almost identical to the original, executing the same code but with its own data space, environment,
and file descriptors.

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

pid_t fork(void);
```

The call to fork in the parent returns the PID of the new child process, zero in the new process.

Example: `new_proc.c`

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

int main()
{
  pid_t pid;
  char *message;
  int n;

  printf("Start\n");
  pid = fork();
  switch(pid){
    case -1:
      perror("fork failed");
      exit(1);
    case 0:
      message = "I am the child";
      n = 7;
      break;
    default:
      message = "I am the parent";
      n = 3;
      break;
  }

  for(; n > 0; n--){
    puts(message);
    sleep(1);
  }
  exit(0);
}
```

Command to compile `new_proc.c`:

```
  gcc -o newproc new_proc.c
```

Default output file from `gcc` is a binary file called `a.out`, to rename the output file to other name (usually same as the source file), use the **`-o`** option follow by the name.

Execute the program and observe the output. Any strange behavior?

Now compile and execute the second version of previous program, `new_proc2.c`, with a few amendments (**bold** lines).

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

int main(){
  pid_t pid;
  char *message;
  int n, exit_code;

  pid = fork();
  switch(pid){
    case -1:
      perror("fork failed");
      exit(1);
    case 0:
      message = "This is the child";
      n = 7;
      exit_code = 9;
      break;
    default:
      message = "This is the parent";
      n = 3;
      exit_code = 0;
      break;
  }

  for(; n > 0; n--){
    puts(message);
    sleep(1);
  }

  if(pid != 0){ //the parent process
    int stat;
    pid_t child_pid;

    child_pid = wait(&stat);
    printf("Child has finished: PID = %d\n", child_pid);

    if(WIFEXITED(stat))
      printf("Child exited with code %d\n", WEXITSTATUS(stat));
    else
      printf("Child terminated abnormally\n");
  }
  exit(exit_code);
}
```

The following macros are used to determine the exit status of a process and the return value or exit code (assume that return value or exit code of a process is stored in `status`).

- `WIFEXITED(status)` returns nonzero (true) if the process exited normally.
- `WEXITSTATUS(status)` returns the process's exit code.

Signal handling

To use signal handling function, include the following header file:

```
#include <signal.h>
```

Signal handling function is

```
int sigaction(int sig, const struct sigaction *act,
              struct sigaction *oldact);
```

The `sig` specifies the signal and can be any valid signal except `SIGKILL` and `SIGSTOP`.

The `sigaction` structure, used to define the actions to be taken on receipt of the signal specified by `sig`, is defined in `signal.h`. It has at least the following members:

```
void (*) (int) sa_handler
sigset_t sa_mask
int sa_flags
```

`sa_handler` is a pointer to a function called when signal `sig` is received. Use the special values `SIG_IGN` and `SIG_DFL` in the `sa_handler` field to indicate that the signal is to be ignored or the handled by default action, respectively.

The `sa_mask` field specifies a set of signals to be added to the process's signal mask[§] before the `sa_handler` function is called. These are the set of signals that are blocked and won't be delivered to the process, to prevent a signal is received before its handler has run to completion.

Functions used to manipulate sets of signals are

```
int sigaddset(sigset_t *set, int signo);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int signo);
```

`sigemptyset` initialises a signal set to be empty.
`sigfillset` initialises a signal set to contain all defined signals.
`sigaddset` and `sigdelset` add and delete a specified signal (`signo`) from a signal set.

If the value of `sa_flags` is set to `SA_RESETHAND`, the signal handler is reset to default action after a signal has been processed.

The `sigaction` function sets the action associated with the signal `sig`.
If `oldact` is not null, `sigaction` writes the previous signal action to the location it refers to.
If act isn't null, the action for the specified signal is set.
Compile and execute the following program, named `sigctrl.c`. Observe the behavior of the program.

[§]The set of signals that a process is currently blocking is called process's signal mask.

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

void yeah(int sig)
{
  printf("YEAH...! I got signal %d\n", sig);
}

int main()
{
  struct sigaction act;

  act.sa_handler = yeah;
  sigemptyset(&act.sa_mask);
  act.sa_flags = SA_RESETHAND;
  sigaction(SIGINT, &act, 0);

  while(1){
    printf("Waiting signal...");
    printf("(Press CTRL+\\ to terminate this program)\n");
    sleep(1);
  }
}
```

The second version of the above program as below, named `sigctrl2.c`.

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

int c=0;

void yeah(int sig)
{
  printf("YEAH...! I got signal %d\n", sig);
  c++;
}

int main()
{
  struct sigaction act, oldact;

  act.sa_handler = yeah;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGINT, &act, &oldcat);

  while(1) {
    printf("Waiting signal...");
    printf("(Press CTRL+\\ to terminate this program)\n");
    if(c==3)
      sigaction(SIGINT, &oldcat, 0);
    sleep(1);
  }
}
```

Compile, execute and observe the behaviour of the program. How does it works?

Signal can be sent to a process by:

1. entering `kill` command in terminal (Example 1 below)
2. using `kill()` system call within a program (Example 2)

Example 1: `sig1.c`

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

int alarm_fired = 0;

void mysighand(int sig)
{
    alarm_fired = 1;
}

int main()
{
  pid_t pid;
  struct sigaction act;

  act.sa_handler = mysighand;
  sigemptyset(&act.sa_mask);
  sigaddset(&act.sa_mask, SIGALRM);
  act.sa_flags = 0;
  sigaction(SIGALRM, &act, 0);

  while(1){
    printf("waiting for signal...send SIGINT to terminate\n");
    pause();
    if(alarm_fired) printf("Ow...I am on fire!\n");
  }
  return 0;
}
```

Compile and execute Example 1, launch another terminal and execute the command **`ps -a`** to check the PID of Example 1.

To send signal to Example 1 (if the PID is 7027), use the `kill` command as below:

```
kill -SIGALRM 7027
```

To terminate Example 1, send interrupt signal: `kill -SIGINT 7027`

Example 2: `sig2.c`

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>

int alarm_fired = 0;

void mysighand(int sig)
{
    alarm_fired = 1;
}

int main()
{
  pid_t pid;
  struct sigaction act;

  act.sa_handler = mysighand;
  sigemptyset(&act.sa_mask);
  sigaddset(&act.sa_mask, SIGALRM);
  act.sa_flags = 0;

  printf("alarm application starting\n");

  pid = fork();
  switch(pid) {
    case -1: /* Failure */
      perror("fork failed");
      return 1;
    case 0: /* child */
      sleep(4);
      kill(getppid(), SIGALRM);
      return 0;
  }

  /* the parent process */
  sigaction(SIGALRM, &act, 0);
  printf("waiting for signal...\n");
  pause();
  if(alarm_fired)
    printf("Ding dong!\n");
  printf("done\n");
  return 0;
}
```

Compile and execute Example 2, observe the output.

Each signal has its own default action. Here are a few examples:

Signal	Description	Default Action
SIGABORT	*Process abort	Terminate
SIGALRM	Alarm clock	Terminate
SIGCHLD	Child process terminated	Ignored
SIGHUP	Hangup (the tty was closed)	Terminate
SIGINT	Terminal interrupt	Terminate
SIGTERM	Termination	Terminate
SIGUSR1	User-defined signal 1	Terminate
SIGUSR2	User-defined signal 2	Terminate

Signals can be caught or ignored excpet the SIGKILL and SIGSTOP signals.
For example, signals SIGINT and SIGTERM can cause your program terminates. However, these signals can be caught and handled in your program.

Example: `sig3.c`

```
#include <sys/types.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>

int alarm_fired = 0;
void mysighand(int sig) {
  if(sig == SIGALRM)
    alarm_fired = 1;
  else if((sig==SIGINT) || (sig==SIGTERM)){
    printf("Housekeeping...be patient.\n");
    sleep(5);
    printf("Done.\n");
    exit(0);
  }
}

int main() {
  pid_t pid;
  struct sigaction act;

  act.sa_handler = mysighand;
  sigemptyset(&act.sa_mask);
  act.sa_flags = 0;
  sigaction(SIGALRM, &act, 0);
  sigaction(SIGINT, &act, 0);
  sigaction(SIGTERM, &act, 0);

  while(1){
    printf("waiting for signal...\n");
    pause();
    if(alarm_fired) printf("Ow...I am on fire!\n");
  }
  return 0;
}
```

Compile and execute `sig3.c`. Send signal `SIGALRM` and then signal `SIGINT` or `SIGTERM` to the program.