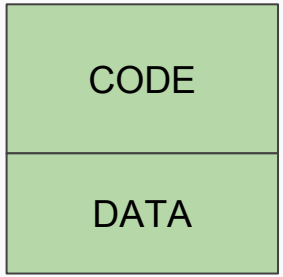
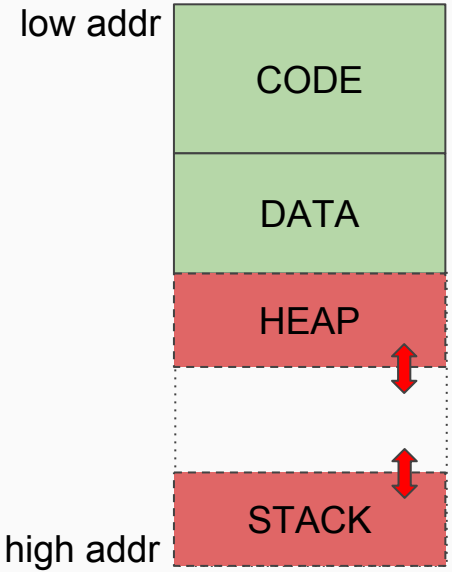
Unix Process Management

Source ⇒ Binary Executable ⇒ Process

```
int main() {
  printf (“Hello”);
  return 0;
}
```





fork()

- Create a new (child) process using the **current copy** of the parent image
- The child image is 99.9% the same as the parent image
- **At the time of fork()**
  - Code section: exact duplicate of each other
  - Heap section: exact duplicate of each other
  - Data/Stack section may differ
- Thereafter, the two images are **independent/unrelated**
  - Parent code and child code **share NOTHING**

Fork(): create a new (child) process whose image is 99.9% the same as the parent

```
/* parent */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

Fork(): create a new (child) process whose image is 99.9% the same as the parent

```
/* parent */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

Fork(): create a new (child) process whose image is 99.9% the same as the parent

a new process is created, DUPLICATING the parent process image

```
/* parent */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

```
/* child */
int main() {
  printf (“Begin\n”);
  fork();
  printf (“PID = %d\n, getpid());
  printf (“End\n”);
  return 0;
}
```

At this point, both parent and child will run independently competing for the same CPU

Fork(): create a new (child) process whose image is 99.9% the same as the parent

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

Fork(): create a new (child) process whose image is 99.9% the same as the parent

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

fork() return value:

- ZERO (in child process)
- child PID (in parent process)

Fork(): create a new (child) process whose image is 99.9% the same as the parent

```
/* parent */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

```
/* child */
int main() {
  printf (“Begin\n”);
  pid_t who = fork();
  if (who == 0)
     printf (“Mug %d\n, getpid());
  else {
     printf (“Cup %d\n, who);
     printf (“Bowl %d\n”, getpid());
  }

  printf (“End\n”);
  return 0;
}
```

exit() & wait()

- `exit()`: terminate and report status to parent
- `wait()`: wait for child to terminate, and accept its status

- ZOMBIE (defunct) processes
  - Child already invoked `exit()`, but parent has not invoked `wait()`
- ORPHAN processes
  - Child is still running, but parent is dead. The child processes will be adopted by the “init” process (PID 1)

Parent-Child Handshake: exit() ⇔ wait()

```
/* parent */
int main() {
  pid_t who = fork();
  if (who == 0) {
     printf (“Mug %d\n, getpid());
     exit (93);
  }
  else {
     int status;
     who = wait (&status);
     printf (“Child status is %d\n”,
        WEXITSTATUS(status));
  }
  return 0;
}
```

```
/* child */
int main() {
  pid_t who = fork();
  if (who == 0) {
     printf (“Mug %d\n, getpid());
     exit (93);
  }
  else {
     int status;
     who = wait (&status);
     printf (“Child status is %d\n”,
        WEXITSTATUS(status));
  }
  return 0;
}
```

`exec*()`: run a new executable

- **Replace** the current process image with a new binary executable
  - Continue running from the “main” of the new executable
- The current process image **stays intact** if the replacement executable cannot be loaded
  - Continue running from the “next” statement in the current process image

`exec*()` variants

- `execl()/execlp()`: the “list” variant
  - (command line) arguments are supplied to the new binary executable using a list
- `execv()/execvp()`: the “vector” variant
  - (command line) arguments are supplied to the new binary executable using an array
- the “p” suffix: use the PATH environment variable to search for the new binary executable
- The FIRST argument to exec* is the **location** of the new binary executable
- The second (and remaining arguments) are arguments passed to the new binary executable

exec*() demo
exec_cal.c

Using exec*()

```
int main() {
   printf (“Begin\n”);
   execl (“/usr/bin/cal”, “Venus”, NULL);
   printf (“End\n”);
   while (1) {}
   return 0;
}
```

when “/usr/bin/cal” replaces the process image,
“End” will never get printed!
the infinite while-loop will never run

```
int main() {
   printf (“Begin\n”);
   execl (“/usr/bin/california”, “Venus”, NULL);
   printf (“End\n”);
   return 0;
}
```

“/usr/bin/california” was not found
the current process image was NOT replaced
“End” will be printed!

exec*() to run “/usr/bin/cal”

```
execl (“/usr/bin/cal”, “Venus”, NULL);        /* list style */
execlp (“cal”, “Venus”, NULL);

char* cmda[] = {“/usr/bin/cal”, NULL};        /* array style */
execv (cmda[0], cmda);
char* cmdb[] = {“cal”, NULL};
execvp (cmdb[0], cmdb);
```

Notes: “Venus” was arbitrarily chosen. It **has “no affect”** on the call, and usually we pass the name of the command (So, “cal” should be used instead of “Venus”)

exec*() to run “/usr/bin/cal -3 2014”

Show 2014 calendar in 3-column format

```
execl (“/usr/bin/cal”, “Venus”, “-3”, “2014”, NULL);
execlp (“cal”, “Venus”, “-3”, “2014”, NULL);

char* cmda[] = {“/usr/bin/cal”, “-3”, “2014”, NULL};
execv (cmda[0], cmda);
char* cmdb[] = {“cal”, “-3”, “2014”, NULL};
execvp (cmdb[0], cmdb);
```