Reading 9

☆Processes

- Process represents an instance of a program running in the system, which includes
  1. Binary executable code
  2. data
  3. execution context
     - ↳ tracks program execution by maintaining its:
       1. register values
       2. stack location
       3. instruction its currently executing

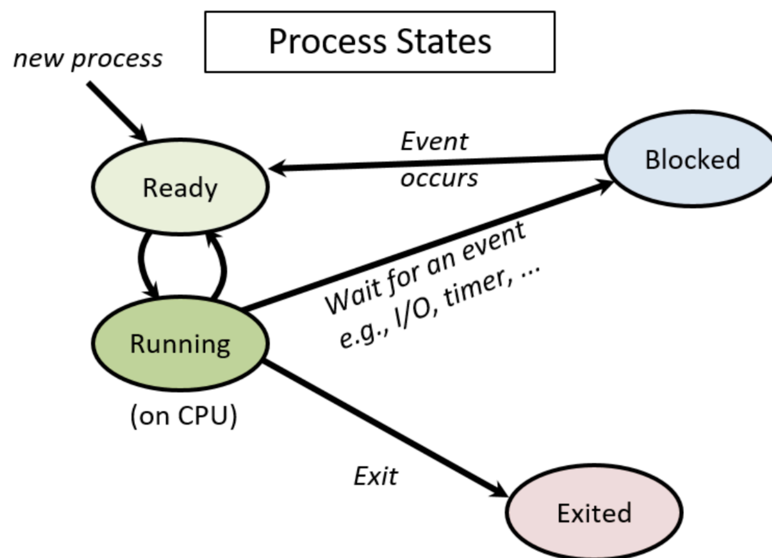
- CPU can only run one thing at a time, so OS is in charge of frequently changing what process is running.

- The policy aspect of multiprogramming governs scheduling the CPU, or picking what process gets to use the CPU and for how long.

- The OS performs context switching, or swapping process state on the CPU as the primary mechanism behind multiprogramming. There are 2 main stages:
  1. Saving the context of the current process running on the CPU
  2. restoring the saved context from another process on the CPU

- OS maintains information about each process including:
  1. PID (process ID)
  2. address space
  3. execution state
  4. set of resources allocated
  5. current process state

- Process States:



- 2 metrics for runtime of process:
  1. Wall time : total duration from ready stage to exit stage (including all stages of process life)
  2. CPU time : Just the time the process spends in running state.

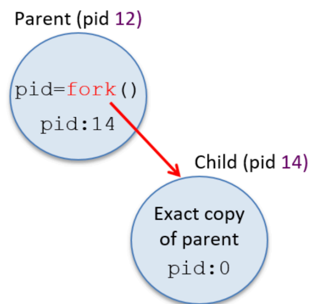
- The 'fork' system call creates a new process. The parent process calls 'fork' and then the child process is created.
  - ↳ The child inherits execution state from parent.

- From the programmers POV, a call to 'fork' returns twice:
  - once in the context of the running parent process
  - once in the context of the running child process

- So a call to 'fork' returns two different values to the parent and child:
  - ↳ child always recieves a return value of 0
  - ↳ parent recieves the child's PID value (or -1 if 'fork' fails)

- Ex

```
pid_t pid;

pid = fork();   /* create a new child process */

print("pid = %d\n", pid);  /* both parent and child execute this */
```



- Ex Application:

```
pid_t pid;

pid = fork();   /* create a new child process */

if (pid == 0) {
    /* only the child process executes this code */
    ...
} else if (pid != -1)  {
    /* only the parent process executes this code */
    ...
}
```

- Remember when you call 'fork' the parent and child processes are concurrent and can be scheduled to run on the CPU in many different orderings, resulting in any possible interleaving of their instruction sequences.

- `Exec()`

    - To initialize the child process to run a new program, the child process calls one of the exec system calls.

    - execvp prototype: `int execvp(char *filename, char *argv[]);`

```
pid_t pid;
int  ret;
char *argv[2];

argv[0] = "a.out";  // initialize command line arguments for main
argv[1] = NULL;

pid = fork();
if (pid == 0) { /* child process */
    ret = execvp("a.out", argv);
    if (ret < 0) {
        printf("Error: execvp returned!!!\n");
        exit(ret);
    }
}
```

- The codes on the left are equivalent because if call to execvp is successful then subsequent lines of code are not executed.

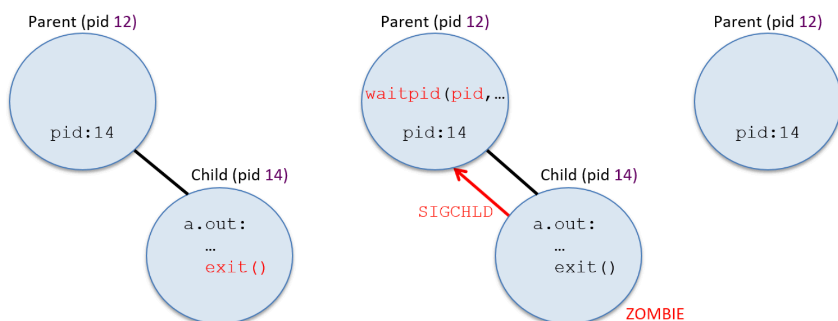
```
pid_t pid;
int ret;

pid = fork();
if (pid == 0) { /* child process */
    ret = execvp("a.out", argv);
    printf("Error: execvp returned!!!\n");  /* only executed if execvp fails *
    exit(ret);
}
```

'Exit()' and 'Wait()'

- To terminate, a process calls 'exit()'
  - ↳ triggers OS to clean up most of process's state
  - ↳ then parent is responsible for cleaning up the exited child's remaining state from the system.

- SIGKILL and SIGINT signals make the process that that recieved the signal call 'exit()'.
  - ↳ The parent recieves SIGCHLD signal to be notified of childs exit.
  - ↳ the child is now a <u>zombie</u>: partially cleaned up but the OS still maintains little info about it.
  - ↳ A parent process <u>reaps</u> its zombie child (cleans up the rest of its state from the system) by calling 'wait()' that takes in a PID arg.

- Sequence of Events:

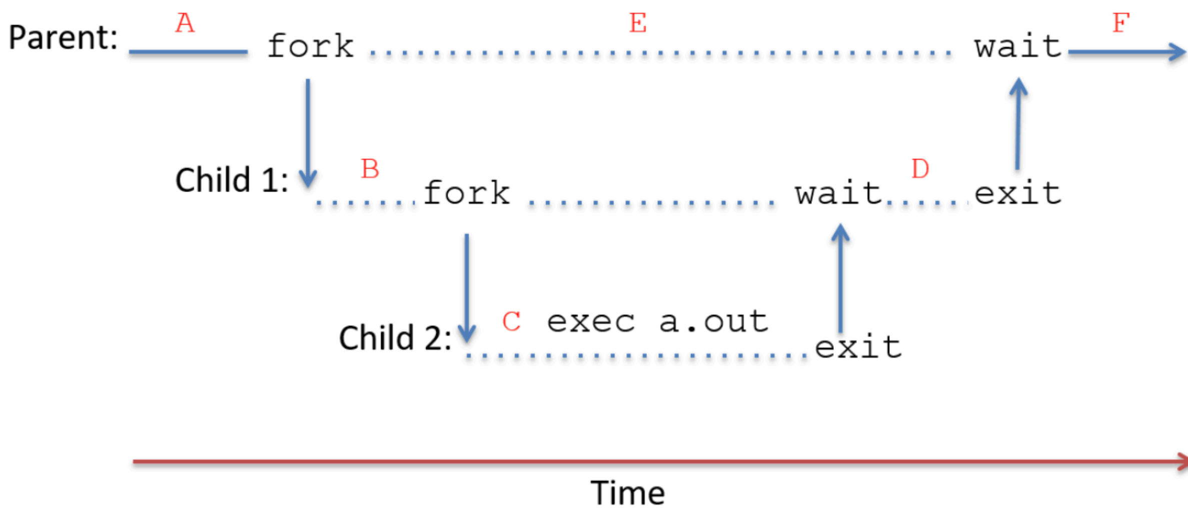


• Ex processes

```c
pid_t pid1, pid2, ret;
int status;

printf("A\n");

pid1 = fork();
if (pid1 == 0 ) {        /* child 1 */
    printf("B\n");

    pid2 = fork();
    if (pid2 == 0 ){     /* child 2 */
        printf("C\n");
        execvp("a.out", NULL);
    } else {             /* child 1 (parent of child 2) */
        ret = wait(&status);
        printf("D\n");
        exit(0);
    }
} else {                 /* original parent */
    printf("E\n");
    ret = wait(&status);
    printf("F\n");
}
```

All Possible Output Orderings from the Program :

Option 1	Option 2	Option 3	Option 4
A	A	A	A
B	B	B	E
C	C	E	B
D	E	C	C
E	D	D	D
F	F	F	F



✩ Signals

- A signal is a software interrupt that is sent by one process to another via the OS.

Signal Name	Description
SIGSEGV	Segmentation fault (e.g., dereferencing a null pointer)
SIGINT	Interrupt process (e.g., Ctrl-C in terminal window to kill process)
SIGCHLD	Child process has exited (e.g., a child is now a zombie after running `exit`)
SIGALRM	Notify a process a timer goes off (e.g., `alarm(2)` every 2 secs)
SIGKILL	Terminate a process (e.g., `pkill -9 a.out`)
SIGBUS	Bus error occurred (e.g., a misaligned memory address to access an `int` value)
SIGSTOP	Suspend a process, move to Blocked state (e.g., Ctrl-Z)
SIGCONT	Continue a blocked process (move it to the Ready state; e.g., `bg` or `fg`)

- Two system calls can be used to change the default behavior of a signal:
  1. sigaction
  2. signal

✩ Process API:

- When 'fork()' is called, the child begins execution from that point in the code, it does NOT start from the beginning of the parent process.

- Remember the parent recieves the PID of child and child recieves zero when 'fork()' returns.

- The output is not deterministic.
  ↳ you don't know whether the parent or child process will run first due to CPU scheduler.

- Parent usually calls 'wait()' to block its execution until child process is done.
  - ↳ adding 'wait()' can make the output deterministic.

- Exec() allows child process to run different code than parent.
  - ↳ given the name of an executable and some arguments, it <u>loads</u> code (and static data) from that executable and overwrites its current code segment (and current static data) with it; the heap and stack and other parts of the memory space of the program are re-initialized.
  - ↳ The OS simply runs that program, passing in any arguments as the argv of that process.
  - ↳ Thus, a new process is <u>NOT</u> created, rather the old process is transformed into the specified program.

- When you run shell commands the shell:
  1. locates executable in file system
  2. Calls fork()
  3. calls some variant of exec()
  4. waits for child to finish by calling wait().

- This somewhat complicated API is used to enable things such as redirecting stdout which is implemented by closing 'stdout' and opening a 'file' in between the 'fork()' and 'exec()' calls.

- Pipes in Unix pipelines are implemented with the 'pipe()' system call which connects the output of one process to an in-kernal pipe and the input of another process is connected to that same pipe.

- The 'kill()' system call is used to send signals to a process.
  - ↳ can be directives to pause, die, or other useful imperatives.

- The 'signal()' system call is used to catch any signals and respond appropriately.

- The superuser of a system is sometimes called root.

ASIDE: KEY PROCESS API TERMS

- Each process has a name; in most systems, that name is a number known as a **process ID** (**PID**).
- The **fork()** system call is used in UNIX systems to create a new process. The creator is called the **parent**; the newly created process is called the **child**. As sometimes occurs in real life [J16], the child process is a nearly identical copy of the parent.
- The **wait()** system call allows a parent to wait for its child to complete execution.
- The **exec()** family of system calls allows a child to break free from its similarity to its parent and execute an entirely new program.
- A UNIX **shell** commonly uses `fork()`, `wait()`, and `exec()` to launch user commands; the separation of fork and exec enables features like **input/output redirection**, **pipes**, and other cool features, all without changing anything about the programs being run.
- Process control is available in the form of **signals**, which can cause jobs to stop, continue, or even terminate.
- Which processes can be controlled by a particular person is encapsulated in the notion of a **user**; the operating system allows multiple users onto the system, and ensures users can only control their own processes.
- A **superuser** can control all processes (and indeed do many other things); this role should be assumed infrequently and with caution for security reasons.