Processes

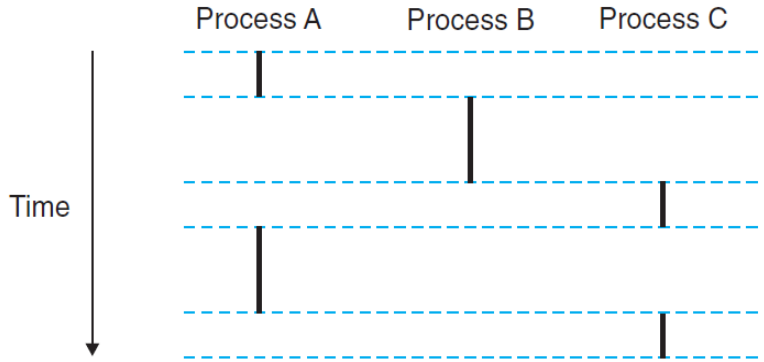
COMPUTER SYSTEMS ORGANIZATION 2021

Process Flows

The key abstractions that a process provides to the application:

- An independent logical control flow that provides the illusion that our program has exclusive use of the processor.
- A private address space that provides the illusion that our program has exclusive use of the memory system.

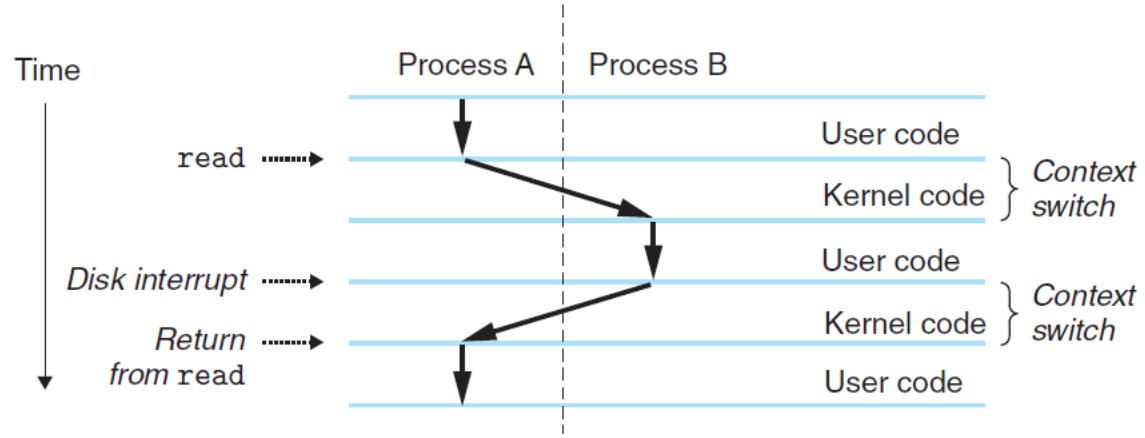
Figure 8.12
Logical control flows. Processes provide each program with the illusion that it has exclusive use of the processor. Each vertical bar represents a portion of the logical control flow for a process.



Each process executes a portion of its flow and then is *preempted* (temporarily suspended) while other processes take their turns.

How is it done ?

- **Private Address Space** - A process provides each program with the illusion that it has exclusive use of the system's address space.
- **User and Kernel Modes** –Restrict the instructions that an application can execute, as well as the portions of the address space that it can access. Use of dual modes User Mode/Kernel Mode during program execution.
- **Context Switch** – Whenever process halts context switch it with another running process.



System Call Error Handling

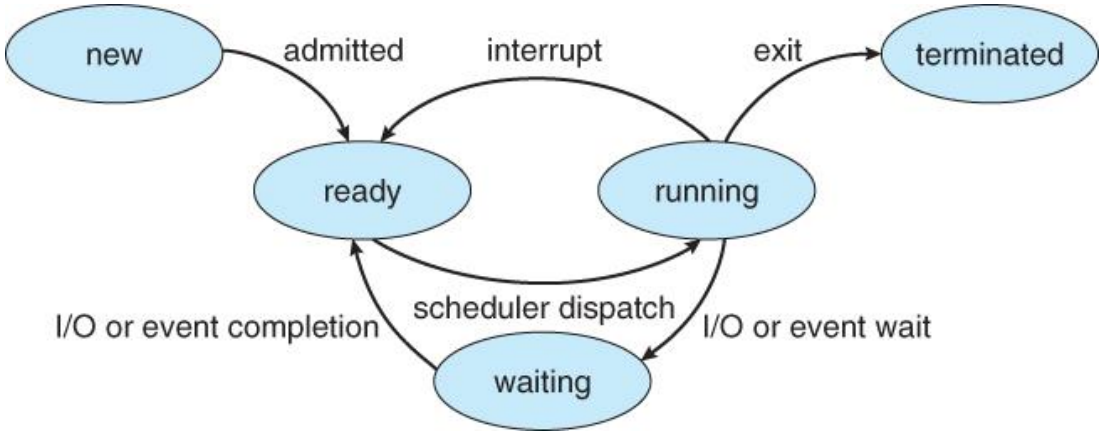
- When Unix system-level functions encounter an error, they typically return −1 and set the global integer variable errno to indicate what went wrong.
- We can simplify our code even further by using error-handling wrappers. pid = Fork();

```
pid_t Fork(void)
{
    pid_t pid;

    if ((pid = fork()) < 0)
        unix_error("Fork error");
    return pid;
}
```

Process Control

- Each process has a unique positive (nonzero) process ID (PID).
- The getpid function returns the PID of the calling process.
- The getppid function returns the PID of its **parent** (i.e., the process that created the calling process).



Life of a process

A parent process creates a new running child process by calling the fork function.

Process Reproduction

- The newly created child process is almost, but not quite, identical to the parent.
- The child gets an identical (but separate) copy of the parent's user-level virtual address space.
- The child also gets identical copies of any of the parent's open file descriptors.
- The most significant difference between the parent and the newly created child is that they have different PIDs.
- The fork function is interesting (and often confusing) because it is called once but it returns twice: once in the calling process (the parent), and once in the newly created child process.
- In the parent, fork returns the PID of the child.
- In the child, fork returns a value of 0.

```c
#include "csapp.h"

int main()
{
    pid_t pid;
    int x = 1;

    pid = Fork();
    if (pid == 0) {  /* Child */
        printf("child : x=%d\n", ++x);
        exit(0);
    }

    /* Parent */
    printf("parent: x=%d\n", --x);
    exit(0);
}
```

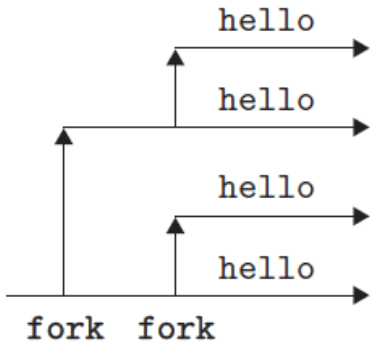
Fork();
Fork();
Fork();

How many processes created ? ANS = 8

(c) Calls `fork` twice

```
1   #include "csapp.h"
2
3   int main()
4   {
5       Fork();
6       Fork();
7       printf("hello\n");
8       exit(0);
9   }
```

(d) Prints four output lines



process graph

Death of your child

- When a process terminates for any reason, the kernel does not remove it from the system immediately.
- Instead, the process is kept around in a terminated state until it is eaten/reaped by its parent.
- A killed child not eaten by its parents becomes a Zombie.
- Zombie exists when a parent gets killed before it could eat its own child.
- In such situation GOD(the init() process) comes to the rescue and kills the zombie.
- The GOD/init process has a PID of 1 and is created by the kernel during system initialization.
- A parent waits for the child to die before eating using the waitpid function.

Process Address Space