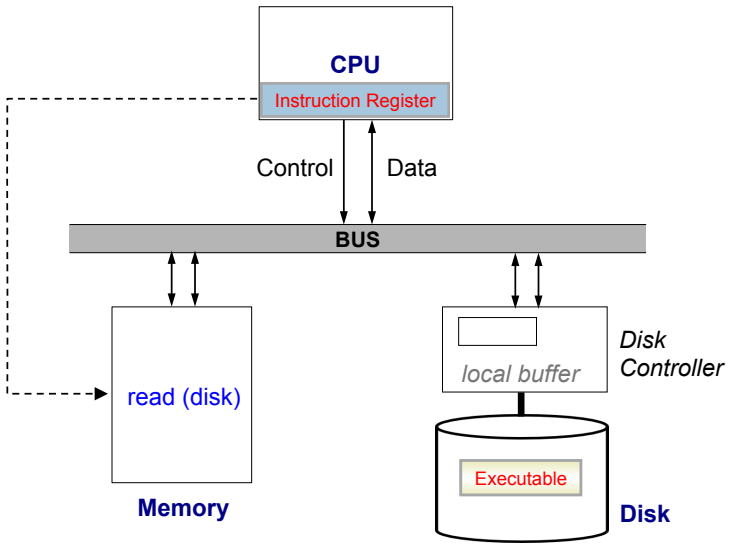
Computer Systems II

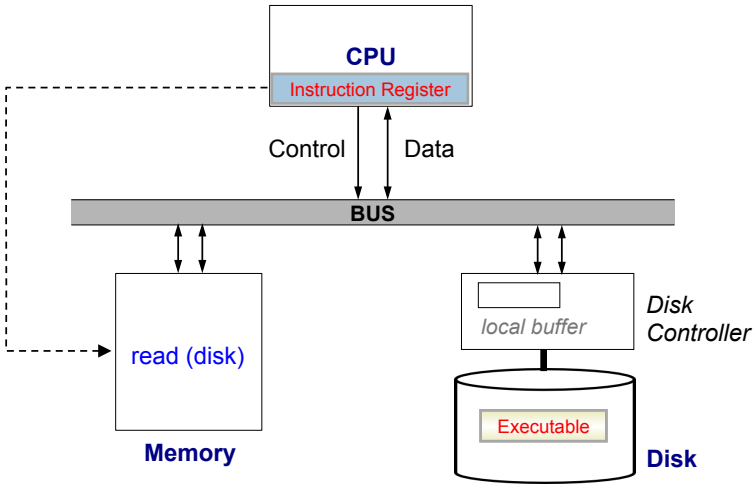
Introduction to Processes

Review: Basic Computer Hardware



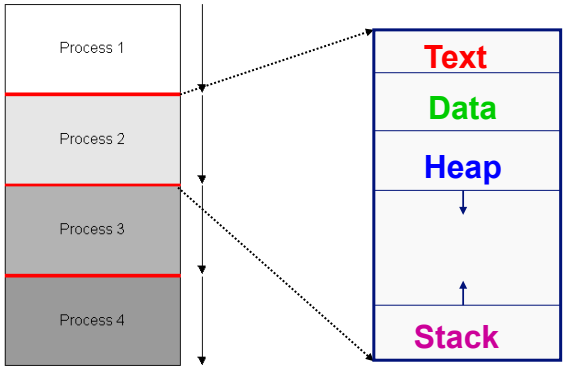
Review: Timing Problem

- I/O devices much slower than the CPU
  - Billions of times slower

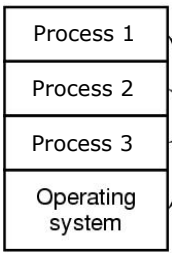


Review: Multiprogramming

- Solution to keeping the CPU busy:
  - Multiprogramming
  - Allow multiple running processes
- When one process blocks, switch to another

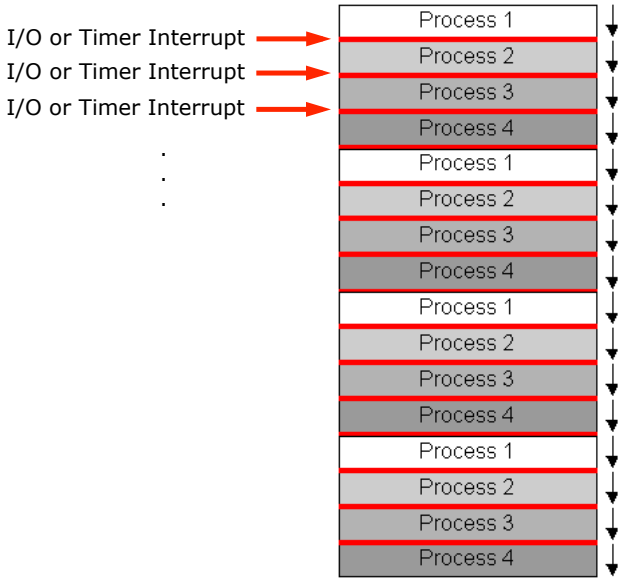


Recall: Multiprogramming Issue



- Processes share the CPU
  - *What if a program has infinite loops?*
- *Solution:* Timer Interrupts
  - The OS sets the internal clock to generate an interrupt at some specified future time
  - This interrupt time is the *process quantum*
  - At each timer interrupt, the CPU switches to another process. This prevents the system being held up by processes in infinite loops.

Multiprogramming with Timer Interrupts

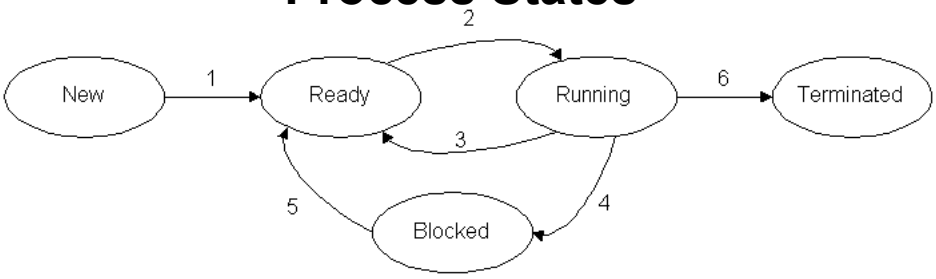


Timer Interrupts Example

- Suppose P1, P2 and P3 are ready to run (in this order). P1 executes for 3 time units, P2 executes for 4 times units, and P2 executes for 2 time units. Neither process invokes I/O. Assuming that a timer interrupt occurs every time unit, fill in the table below:

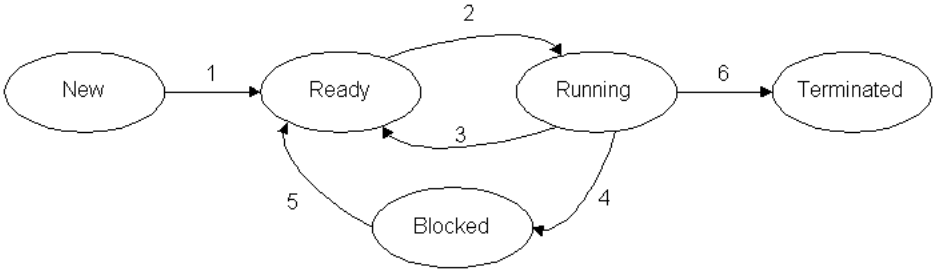
Time (in units)	Running	Queue of Processes Ready to Run
1+ε		
2+ε		
3+ε		
4+ε		
5+ε		
6+ε		
7+ε		
8+ε		
9+ε		

Process States



- Process States
  - (New) Process is being created
  - (Ready) Process is waiting its turn to use the CPU
  - (Running) Process is actively using the CPU
  - (Blocked) Process is waiting for I/O to complete or for some other event to happen
  - (Terminated) Process has finished execution

Process State Transitions



- State Transitions
  1. Process created, placed into Ready queue
  2. The OS has selected a process to run
  3. Timeout: the process has used up its allotted time slice
  4. Need I/O or need to wait until a future event
  5. I/O done or event occurred
  6. Process has finished execution or has been terminated

State Transition Example

- Suppose P2, P3, P4 are in this order in the Ready Queue, and that the following sequence of events occurs:

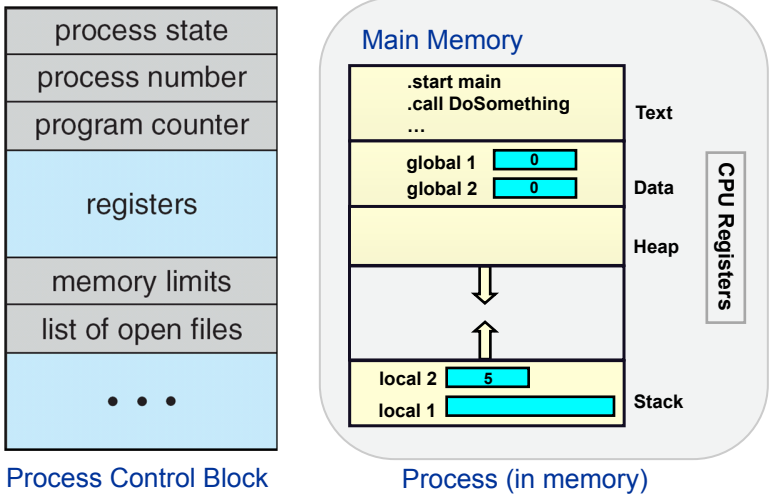
Time 5: P1 invokes read
Time 15: P2 invokes write
Time 20: Interrupt (P1's read complete)
Time 30: P3 terminates
Time 35: Interrupt (P2's write complete)
Time 40: P4 terminates
Time 45: P1 terminates

Time	Running	Ready	Blocked
5+ε			
15+ε			
20+ε			
30+ε			
35+ε			
40+ε			
45+ε			

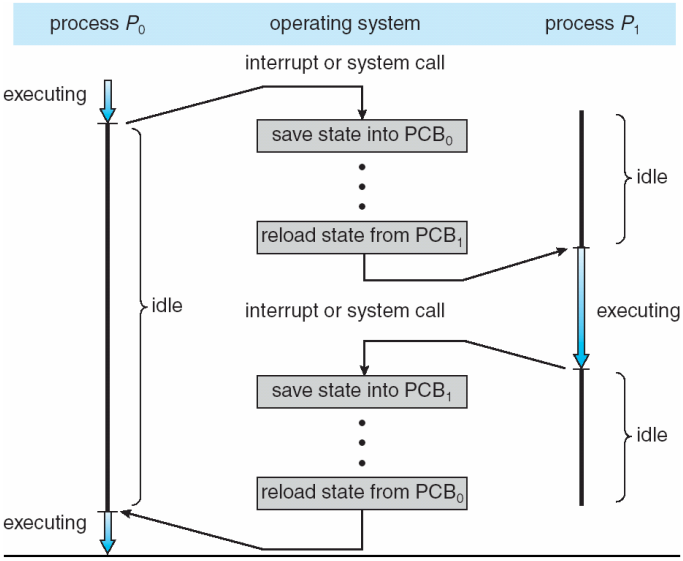
Assume that processes are always appended at the *end* of a queue, and the *first* process in the queue is always scheduled. Fill in the table to show the process states after each event.

Process Control Block (PCB)

- Operating system data structure
  - maintains information associated with each process (context)



CPU Switch From Process to Process

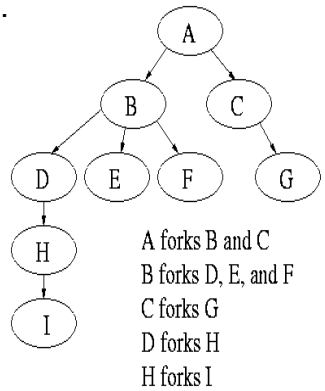


Process API

`fork`
`wait`
`exit`

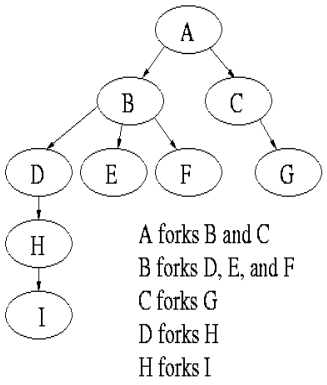
Creating Processes: fork

- Modern general purpose operating systems permit a user to create and destroy processes.
- In Unix this is done by the **fork** system call, which creates a **child** process, and the **exit** system call, which terminates the current process.



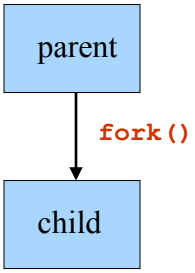
Process Tree

- After a **fork**, both parent and child keep running, and each can fork off other processes.
- A process tree results. The root of the tree is a special process created by the OS during startup.
- A process can *choose* to wait for children to terminate. For example, if C issued a **wait**() system call, it would block until G finished.



How fork Works

- A process invokes fork to create a child process
- Parent and child execute concurrently
- Child process is a duplicate of the parent process

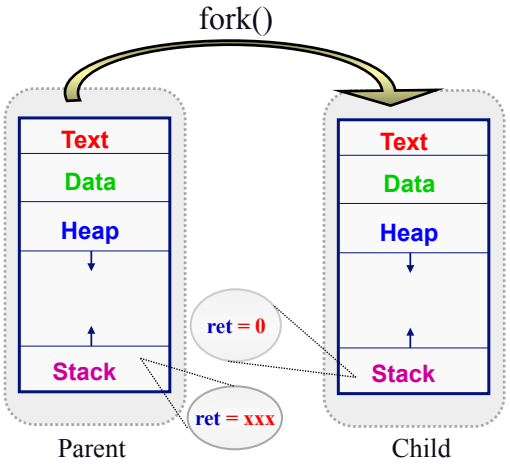


fork System Call

```
ret = fork();
```

- Calling process split into two: parent, child

- ret = -1 if unsuccessful
- ret = 0 in the child
- ret = the child's identifier in the parent



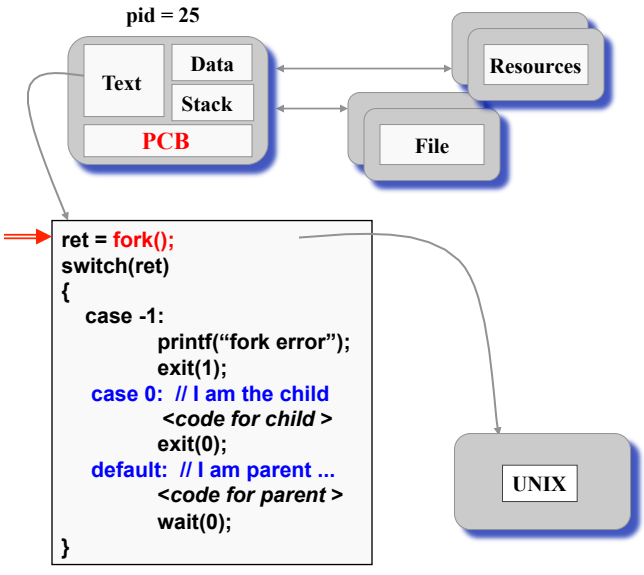
Try It Out

```
#include <sys/types.h>
#include <unistd.h>
#include <stdio.h>
int main()
{
    pid_t  ret;
    pid_t myid;

    /* fork another process */
    ret = fork();
    if (ret < 0) { /* error occurred */
        printf("Fork Failed");
        return 1;
    }

    myid = getpid();
    printf("ret = [%d], myid = [%d]\n", ret, myid);

    return 0;
}
```

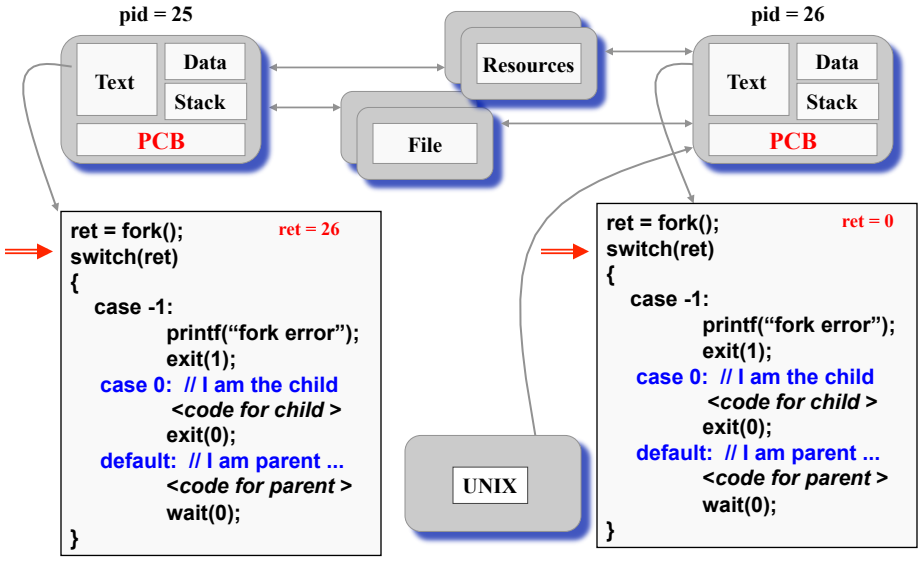
fork System Call

- The child process inherits from parent
  - identical copy of memory
  - CPU registers
  - all files that have been opened by the parent
- Execution proceeds concurrently with the instruction following the fork system call

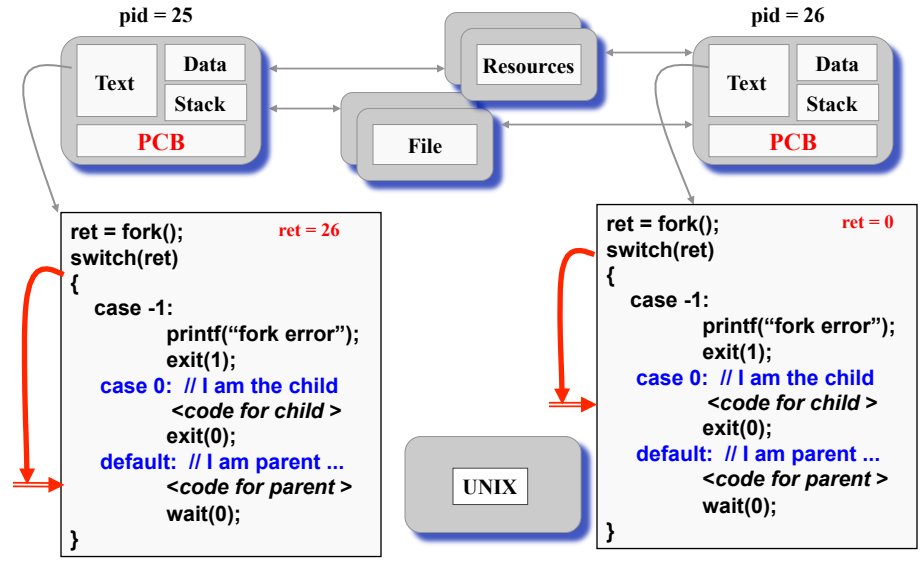
How fork Works



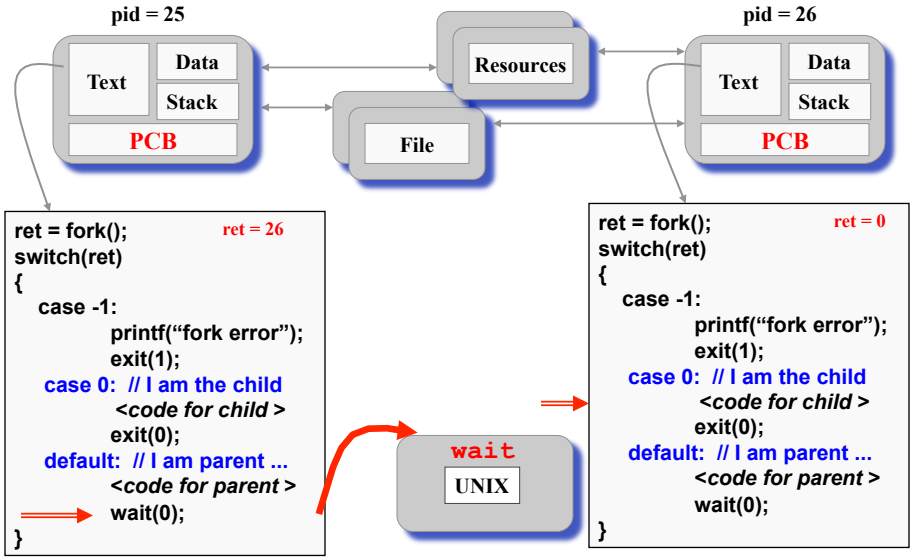
How fork Works



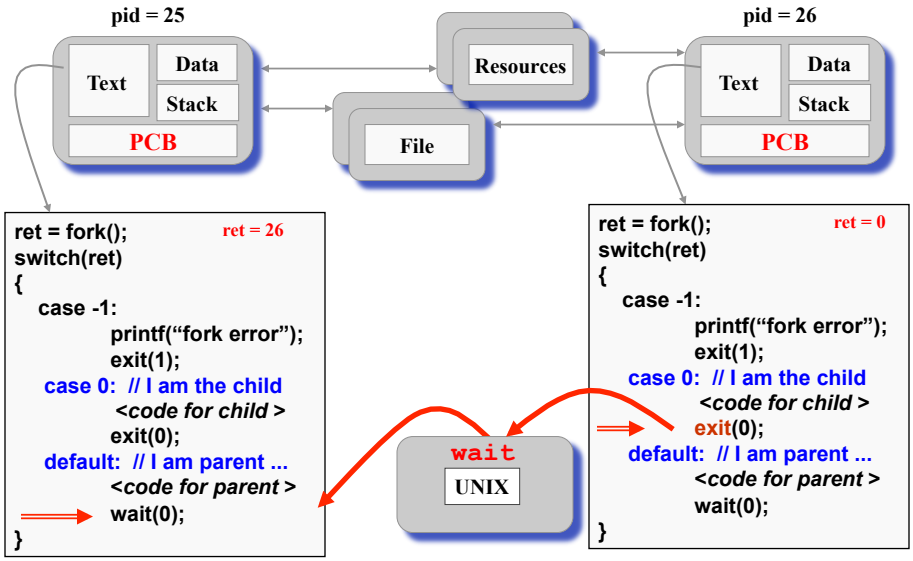
How fork Works



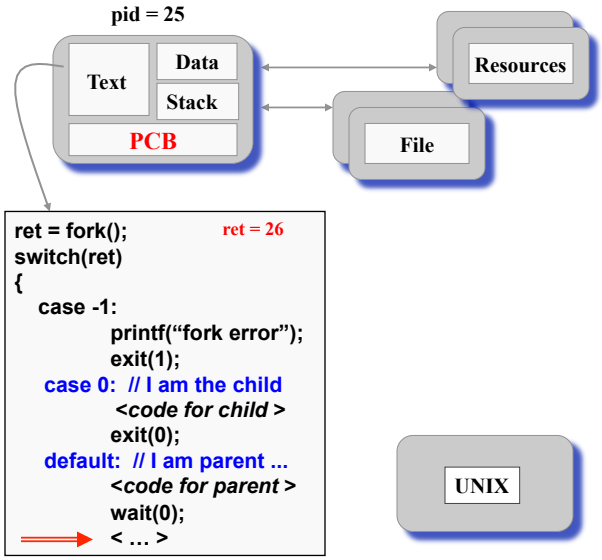
How wait Works



How exit Works



How fork Works (6)



Orderly Termination: exit()

- To finish execution, a child may call exit(*number*)
- This system call:
  - Saves result = argument of exit
  - Closes all open files, connections
  - Deallocates memory
  - Checks if parent is alive
  - If parent is alive, holds the result value until the parent requests it (with wait); in this case, the child process does not really die, but it enters a zombie/defunct state
  - If parent is not alive, the child terminates (dies)

Notes

- Zombies can be noticed by running the `ps` command (shows the process list); you will see the string `<defunct>` as their command name:

```
ps -ef
ps -ef | grep mdamian
```

- The first column of the result is the process identifier
- To force a process to finish executing, use kill:

```
kill -s KILL processid
(or)  kill -SIGKILL processid
(or)  kill -9 processid
```

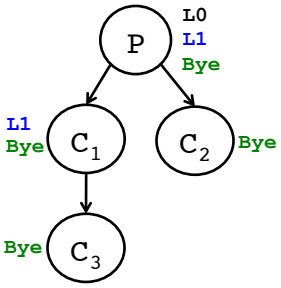
Fork Example 1: What Output?

```
int x = 1;

int main()
{
   pid_t ret;

   ret = fork();
   if (ret != 0) {
     printf("parent: x = %d\n", --x);
     exit(0);
   } else {
     printf("child: x = %d\n", ++x);
     exit(0);
   }
}
```

Fork Example 2

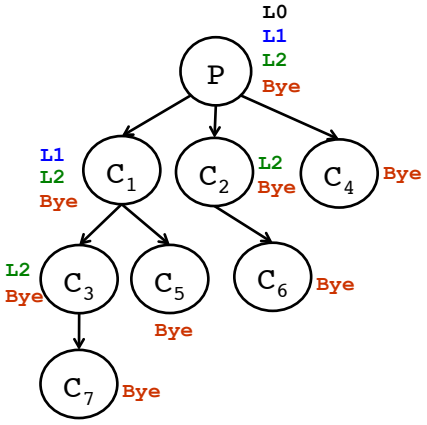
- Key Points
  - Both parent and child can continue forking

```
void fork2()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("Bye\n");
}
```



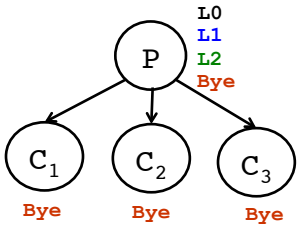
Fork Example 3

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```



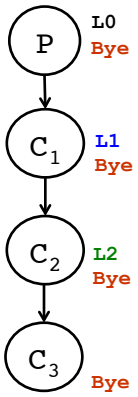
Fork Example 4

```
void fork4()
{
    printf("L0\n");
    if (fork() != 0) {
       printf("L1\n");
       if (fork() != 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Fork Example 5

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```



Waiting for the Child to Finish

- Parent may want to wait for children to finish
  - Example: a shell waiting for operations to complete
- Waiting for any some child to terminate: wait()
  - Blocks until some child terminates
  - Returns the process ID of the child process
  - Or returns -1 if no children exist (i.e., already exited)
- Waiting for a specific child to terminate: waitpid()
  - Blocks till a child with particular process ID terminates

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

pid_t wait(int *status);
pid_t waitpid(pid_t pid, int *status, int options);
```

Other Useful System Calls

- exit terminates the execution of the calling process:

```
exit(0);
```

- getpid returns the identifier of the calling process. Example call (pid is an integer):

```
pid = getpid();
```

- getppid returns the identifier of the parent.

Hands On

- Write a C program that does the following:
  - Takes a list of file names from the command line

```
./a.out readme code.cpp game.input
```

  - For each file in the list:
    - Parent prints out the filename, forks child, waits for child

```
readme
code.cpp
game.input
```

    - Child displays the contents of the file (use `tail`)

```
invoke system("tail readme")
invoke system("tail code.cpp")
invoke system("tail game.input")
```

Hands On (contd.)

- After all children finished executing, the parent prints out

N files processed, done!

where N is the number of filenames in the command line.

Summary

- Process
  - program in execution
- Multiprogramming
  - processes executing concurrently
  - take turns using the CPU
- Process Control Block
  - data structure used by the OS to track processes
- Creating and synchronizing processes
  - system calls fork, wait, exit