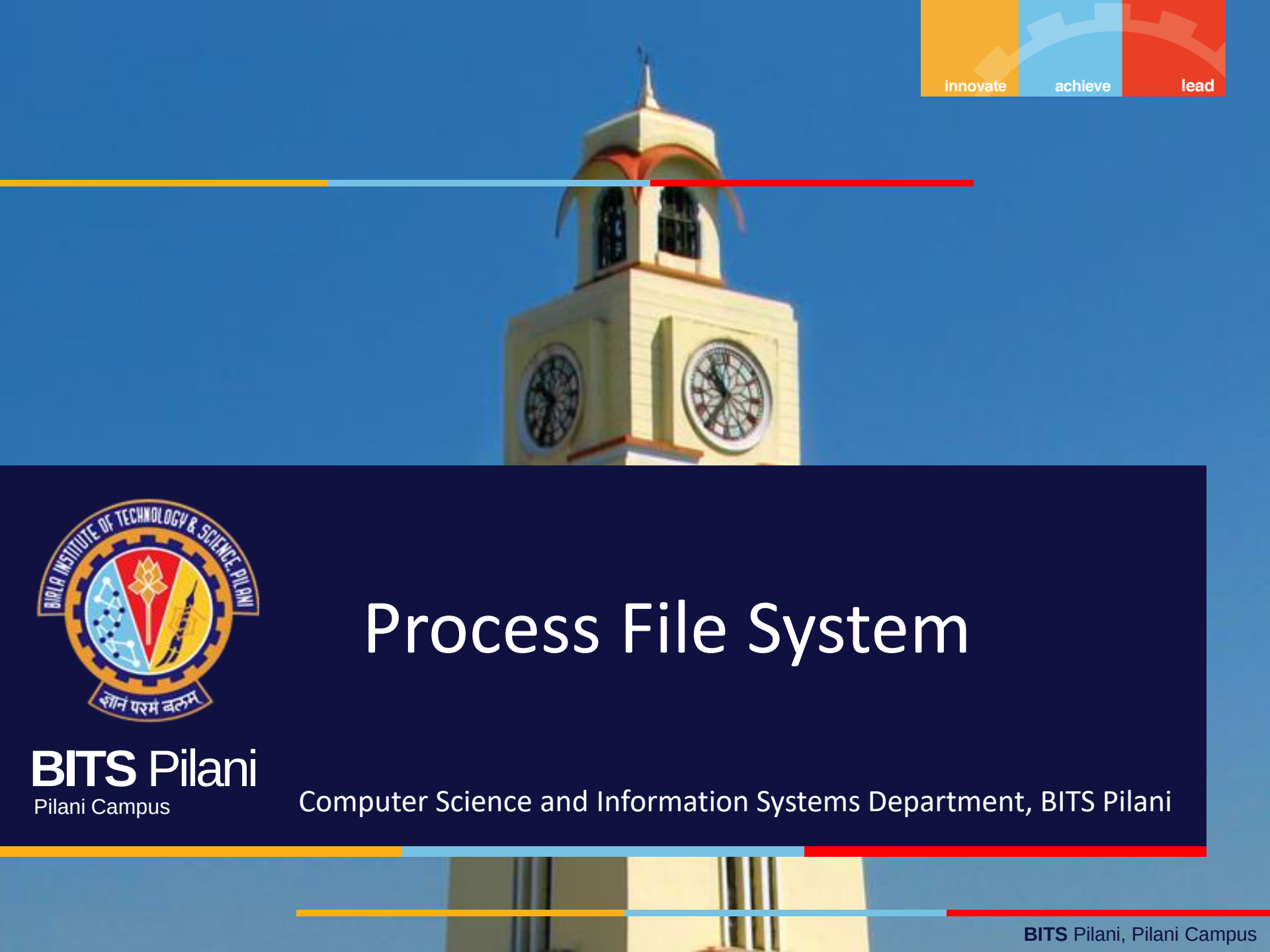


innovate achieve lead

Process File System

BITS Pilani
Pilani Campus

Computer Science and Information Systems Department, BITS Pilani

Today's Agenda

- Unix System Calls
  - Process Creation
  - Process Execution

Process

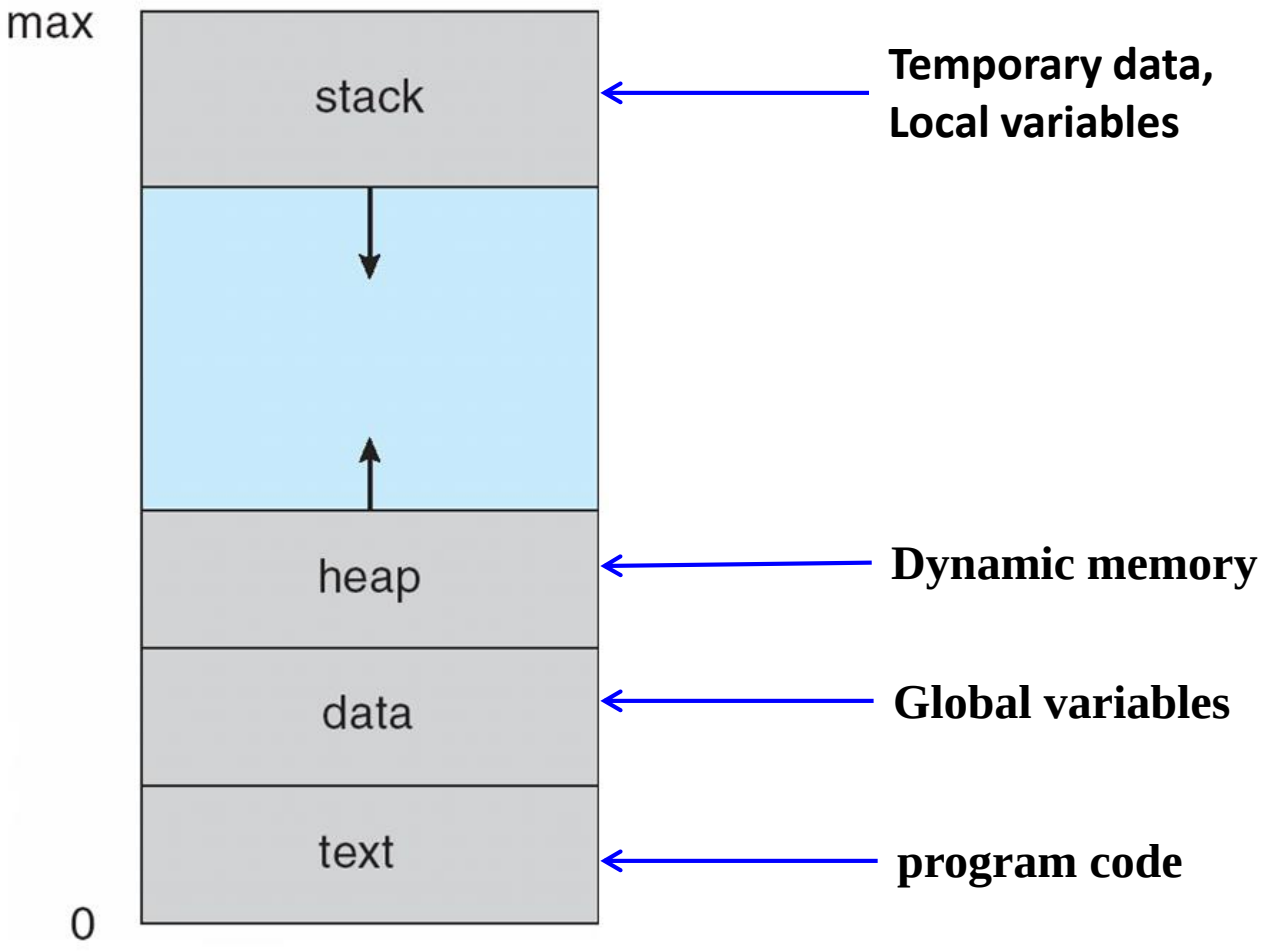
- Process is an instance of executing program
- Main function of process is to *Execute Instructions* residing in main memory
- Process is characterized by
  - Its code, Data, stack and set of register
- During its life time, it can be in different states

  such as running, not running

The Process

- Includes
  - The program **code**, also called text section
  - Current activity including value **of program counter** and **processor registers**
  - **Stack** containing temporary data
    - Function parameters, return addresses, local variables
  - **Data** section containing global variables
  - **Heap** containing memory dynamically allocated during run time

Process in Memory



Local variables (inside function) vs. Global variable (outside function)

Process Creation

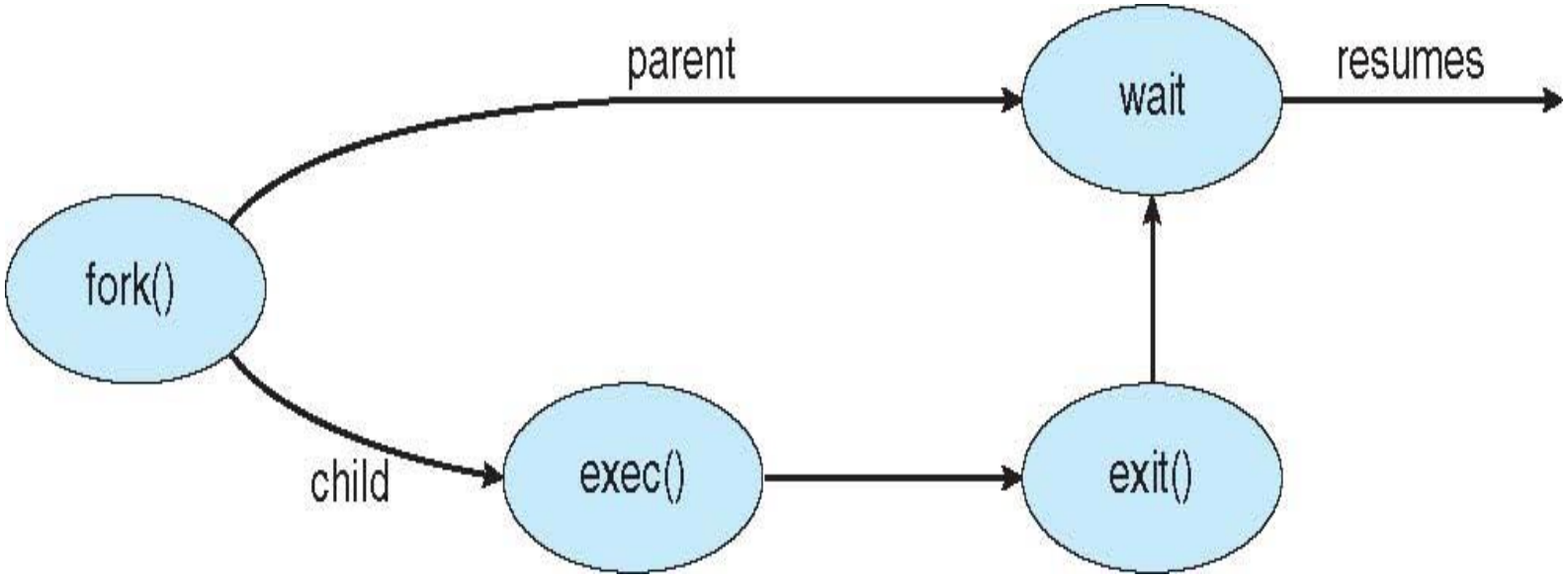
- **Parent** process create **children** processes, which, in turn create other processes, forming a tree of processes
- Generally, process identified and managed via **a process identifier** (**pid**)
- **Resource sharing**
  - Parent and children share all resources
  - Children share subset of parent's resources
- **Execution**
  - Parent and children execute concurrently
  - Parent waits until children terminate

Process Creation (Cont.)

- Address space
  - Child process is duplicate of parent process
  - Child process has a program loaded into it

- UNIX examples
  - **fork** system call creates new process
  - **exec** system call used after a **fork** to replace the process' memory space with a new program

Process Creation (Cont.)



getpid() & getppid() system calls in Unix

- To obtain current process id, use **getpid() system call.**

- To obtain parent process id, use **getppid() system call.**

Example

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

int main() {
printf("I am process %ld\n",(long)getpid());
printf("My Parent is %ld\n",(long)getppid());
}
```

```
output
csis@csis-Latitude-E5450:~$ ./a.out
I am process 3193
My Parent is 2981
```

Fork() and exec() system calls

- **Fork()**
  - It creates a new process which is an identical copy of an existing process.
  - The newly created process will contain all the instructions and data of its parent process.
  - Hence it executes the same parent process.

Fork() and exec() system calls (cont.)

- **Exec()**
  - This on the other hand re-initializes the existing process with some other designated program.
  - It does not create a new process.
  - It merely flushes the current context of a program and loads a new context (new program).
  - exec() call is the only way to execute programs in UNIX. In fact, the kernel boots itself using the exec() call.
  - fork() is the only way to create new processes in UNIX

fork() system call

- fork() call creates a “new” process.
- The child process’ context will be the same as the parent process.
- After a fork() call, two copies of the same context exist, one belonging to the child and another to the parent.
- Contrast this to exec(), where a single context will exist because of child context over-writing the parent.

```
# include<unistd.h>
int fork(void);
/* On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created.*/
```

Example:

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main(void)
{
      printf("Hello \n");
      fork();
      printf(“Bye\n");
      return 0;
}
```

Output:

Hello
Bye
Bye

```c
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
main() {
        int pid;
        pid = fork();
        if (pid < 0) { // error occurred
                fprintf(stderr, "Fork failed!\n");
                exit(-1); }
        else if (pid == 0) { // child process
                printf("I am the child, return from fork=%d\n", pid);
                execlp("/bin/ps", "ps", NULL); }
        else { // parent process
                printf("I'm the parent, return from fork, child pid=%d\n", pid);
                printf("Parent exiting!\n");
                exit(0); }
}
```

Output:

```
programcreek:~/cdtworkspace/CPractice/src> ./a
I’m the parent, return from fork, child pid=15053
Parent exiting!
I am the child, return from fork=0
programcreek:~/cdtworkspace/CPractice/src>   PID TTY      TIME CMD
15033 pts/      0 00:00:00 tcsh
15053 pts/      0 00:00:00 ps
```

Fork() with exec() system call

```
1)  int main(int argc, char *argv[]){
2)      int pid;
3)      printf("*****before fork****\n");
4)      system("ps");
5)      pid = fork();
6)      if(pid == 0)
7)      {
8)          printf("*****after fork****\n");
9)          printf("child pid = %d\n",pid);
10)     }
11)     else
12)     {
13)         printf("parent pid = %d\n",pid);
14)         wait();
15)     }
16)  return 0;
17)  }
```

output

```
[csis@localhost processes]$ ./a.out
*****before fork*****
  PID TTY          TIME CMD
 2806 pts/0    00:00:00 bash
 4196 pts/0    00:00:00 a.out
 4197 pts/0    00:00:00 ps
 parent pid = 4198
 *****after fork*****
 child pid = 0
```

Multiple Fork System Call

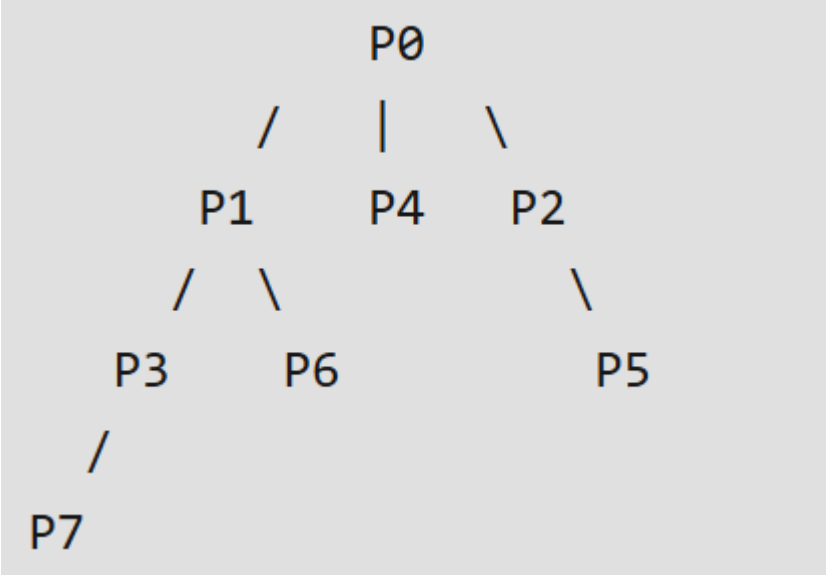
```
#include <stdio.h>
#include <sys/types.h>
int main()
{
   fork();
   fork();
   fork();
   printf("hello\n");
   return 0;
}
```

Multiple Fork System Call

```
fork ();    // Line 1
fork ();    // Line 2
fork ();    // Line 3

        L1          // There will be 1 child process
     /     \        // created by line 1.
   L2       L2      // There will be 2 child processes
  /  \     /  \     //  created by line 2
L3   L3   L3   L3   // There will be 4 child processes
                    // created by line 3
```

Multiple Fork System Call



output

S. no.	Fork() system call in program	Print output “Hello”	Parent process	Child Process
1	Fork()	2	1	1
2	Fork() Fork()	4	1	3
3	Fork() Fork() Fork()	8	1	7

Multiple Fork System Call

```
#include <stdio.h>
#include <unistd.h>
int main()
{
   if (fork() || fork())
      fork();
   printf("1 ");
   return 0;
}
```

Output:- 1 1 1 1 1

Multiple forks

```
#include <stdio.h>
#include <unistd.h> /* contains fork prototype */
main(void)
{
    printf("Here I am just before first forking \n");
    fork();
    printf("Here I am just after first forking \n");
    fork();
    printf("Here I am just after second forking \n");
    printf("\t\tHello World from process %d!\n", getpid());
}
```

output

```
Here I am just before first forking
Here I am just after first forking
Here I am just after first forking
Here I am just after second forking
        Hello World from process 1817!
Here I am just after second forking
Here I am just after second forking
        Hello World from process 1819!
Here I am just after second forking
        Hello World from process 1820!
        Hello World from process 1822!
```

Any Queries?