Chapter 3: Processes (Part 2)

Chapter 3: Processes

- Process Concept
- Process Scheduling
- **Operations on Processes**
- Interprocess Communication
- Examples of IPC Systems
- Communication in Client-Server Systems

Operations on Processes

- There are three commonly seen operations:
  - **Process Creation:** Creates a new process. The newly created is the child of the original. Unix uses ***fork()*** system call to create new processes.
  - **Process Termination:** Terminate the execution of a process. Unix uses ***exit()***.
  - **Process Join:** Wait for the completion of a child process. Unix uses ***wait()***.

- ***fork()***, ***exit()*** and ***wait()*** are system calls.

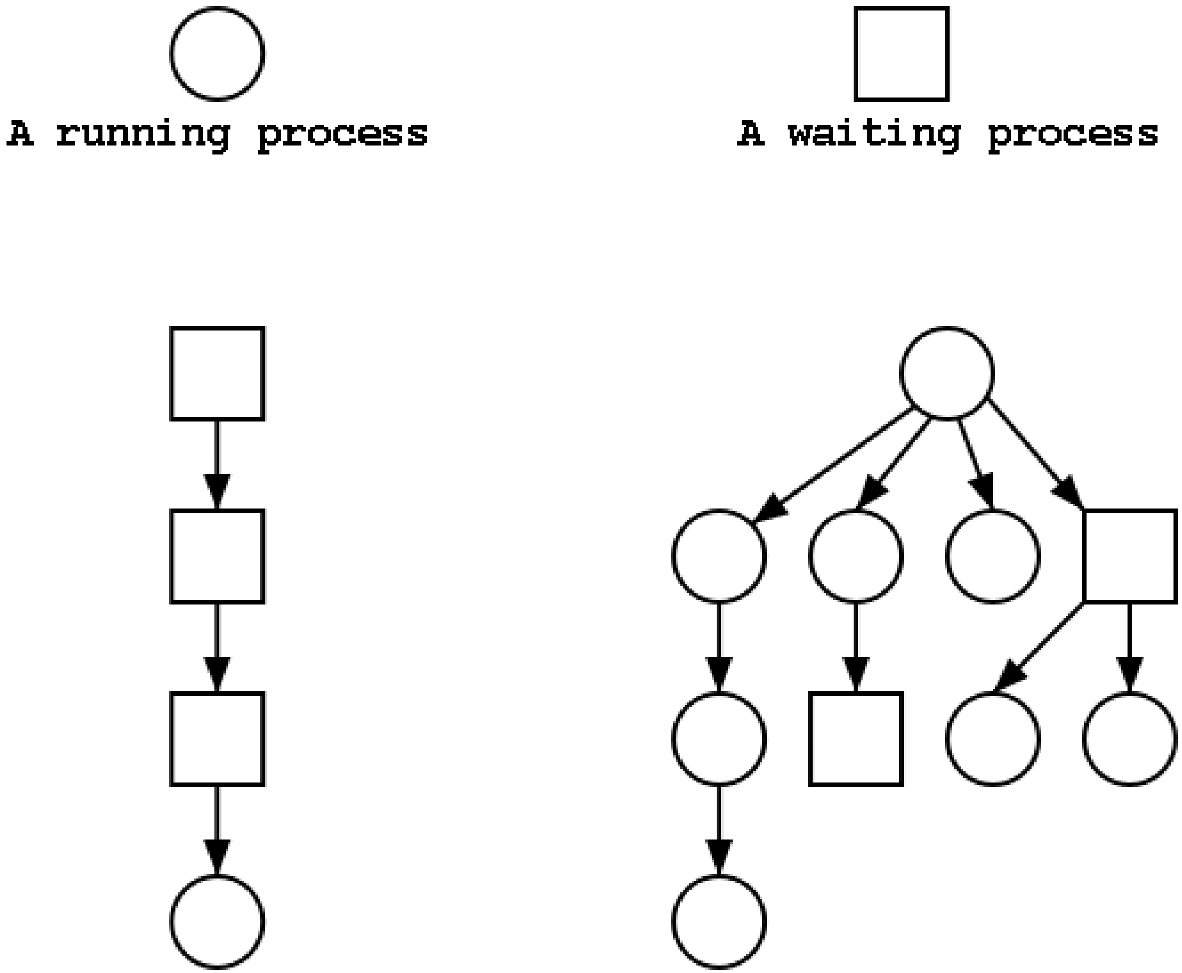
Process Creation

- **Parent** process creates **children** processes, which, in turn create other processes, forming a **tree** of processes
- Generally, process identified and managed via a **process identifier** (**pid**)
- Resource (files etc.) sharing **options**
    1. Parent and children share all resources
    2. Children share subset of parent's resources
    3. Parent and child share no resources
- Execution **options**
    1. Parent and children execute concurrently
    2. Parent waits until children terminate
- Address space options
    1. Child process is duplicate of the parent process (same program and data)
    2. Child process has a new program loaded into it.

Process Hierarchy

- Whilst the operating system can run many processes at the same time, in fact it only ever directly starts one process called **the init** (short for initial) process. This isn't a particularly special process except that it's **PID is always 1** and it will always be running.
  - **Init** is started by the kernel during the booting process;
- All other processes can be considered children of this **initial process**. Processes have a family tree just like any other; each process has a parent and can have many siblings, which are processes created by the same parent.
- Certainly children can create more children and so on and so forth.
- The term **spawn** is often used when talking about parent processes creating children; as in "**the process spawned a child**".
- **pstree** is a linux command that shows running processes as a tree

Process Hierarchy

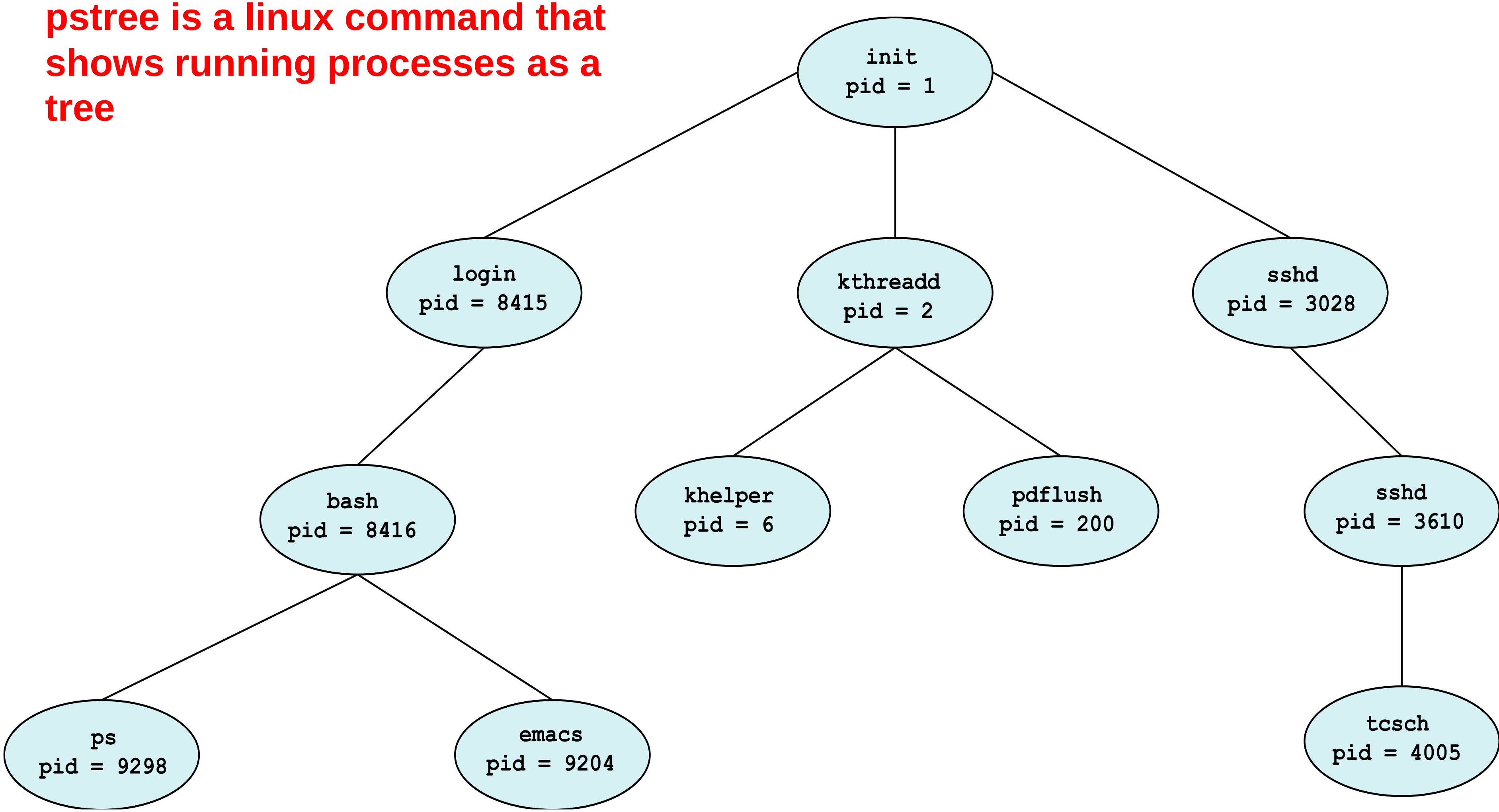


Parent waits for child
Uniprogrammed

Parent free
Multiprogrammed

A Tree of Processes in Linux

pstree is a linux command that shows running processes as a tree



The fork() System Call

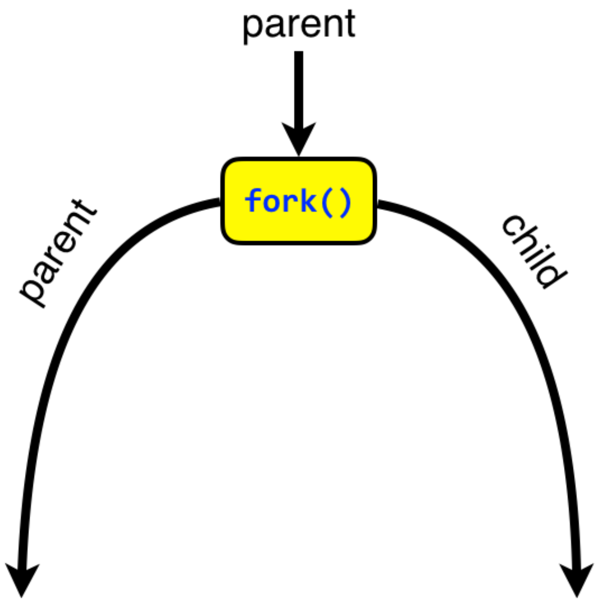
- The purpose of ***fork()*** is to create a child process. The creating and created processes are the ***parent*** and ***child***, respectively.
- Unix will make an exact copy of the parent's address space and give it to the child. Therefore, the parent and child processes have separate address spaces.
- ***fork()*** does not require any argument!
- If the call to ***fork()*** is successful, Unix creates an identical but separate address space for the child process to run.
- Both processes start running with the instruction following the ***fork()*** system call.

Fork() system call

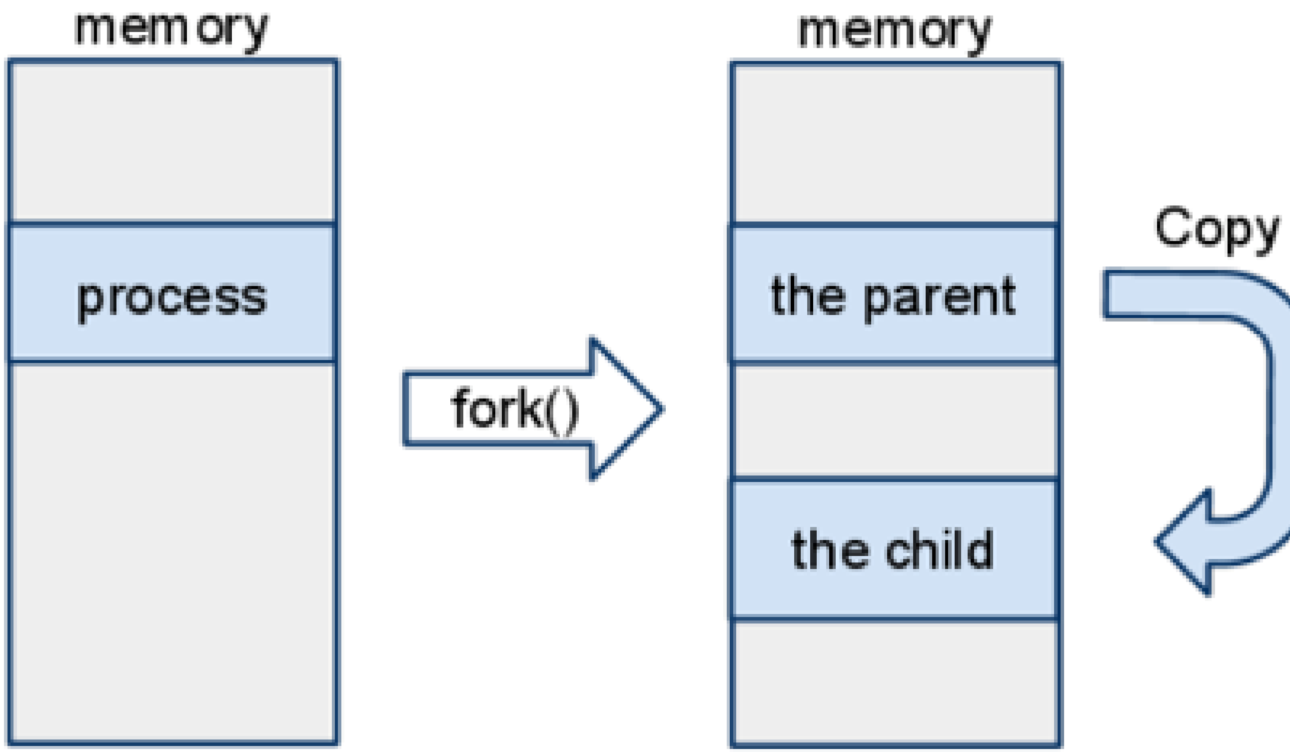
- Child process inherits
  - Stack
  - Memory
  - Open file descriptors
  - Current working directory
  - Resource limits
  - Root directory
- Child process does not inherit
  - Process ID
  - Different parent process ID.

The fork() System Call

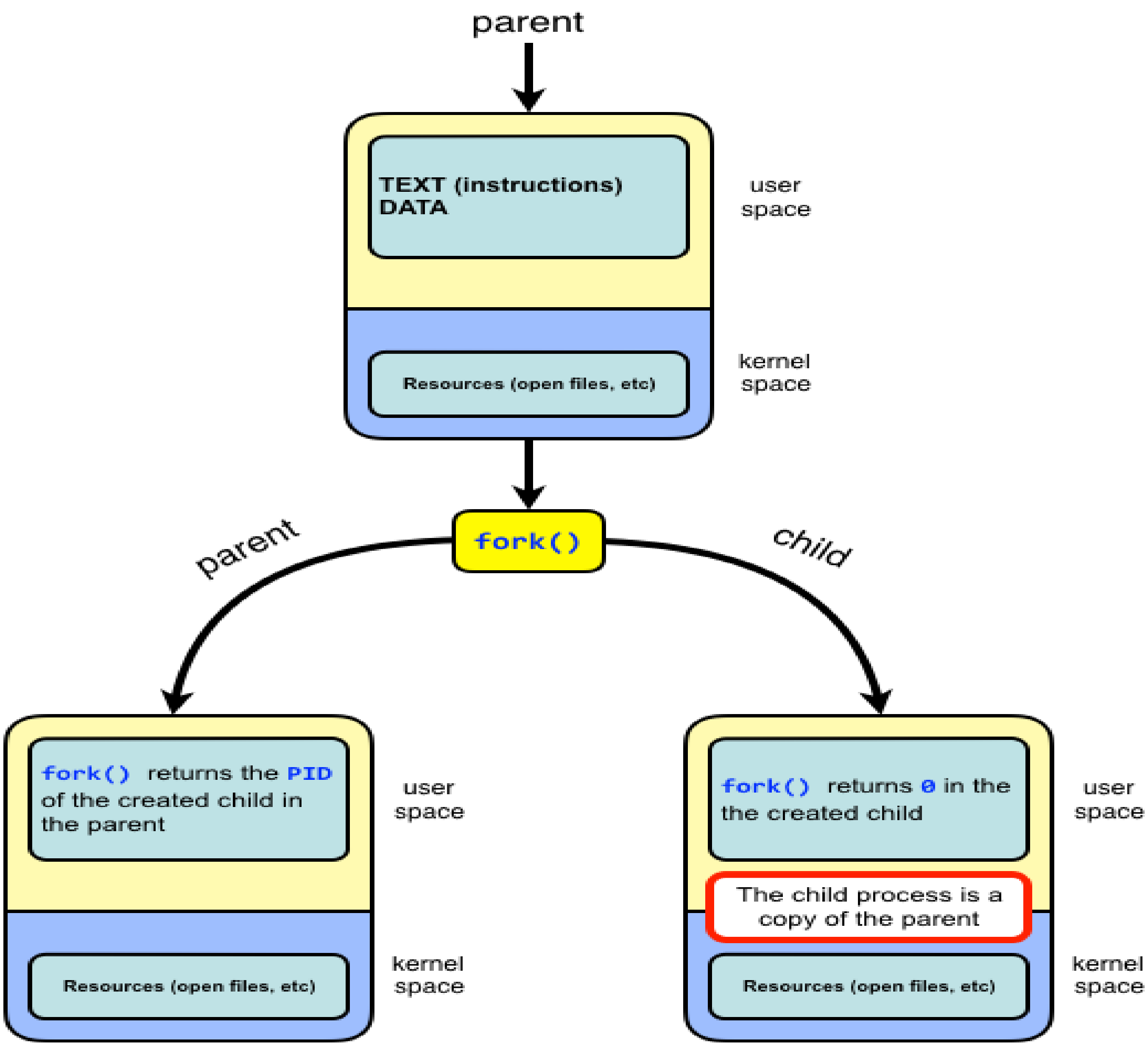
- The process invoking fork is called the **parent**. The new process created as the result of a fork is the **child** of the parent.
- Return Value:
  - **On success,** the PID of the child process is returned in the parent, and 0 is returned in the child.
  - **On failure,** -1 is returned in the parent, no child process is created.



Before and after fork()



The fork() System Call



The fork() System Call

- **On success fork returns twice:**
  - once in the parent and once in the child. After calling fork, the program can use the **fork()** **return value** to tell whether executing in the parent or child.
- If the return **value is 0** the program executes in the **new child process**.
- If the return value is **greater than zero**, the program executes in the **parent process** and the return value is the process ID (PID) of the created **child process**.
- On failure fork returns **-1**.
  - OS does not have enough resources to create a child process

Example 1: 1/2

This example does not distinguish parent and the child processes.

```
#include  <stdio.h>
#include  <sys/types.h>
int  main(void) {
    pid_t  pid;                    //variable pid of type pid_t is declared.
                        //The pid_t data type is the data type used for process IDs.
    int    i;

fork(); //creates a child process. Both processes execute same copies of the below code.
    pid = getpid();            //The value of pid will be different in both child and parent
    for (i = 1; i <= 200; i++) {
  printf("This line is from pid %d, value = %d\n", pid, i); //printf is same as cout in cpp
}
    return 0;
}
```

Example 1: Parent and child execute same copies of code. 2/3

Parent

```
fork();
pid = getpid();
for (i = 1; i <= 200; i++) {
printf("This line is from pid %d, value = %d\n", pid, i);
}
return 0;
}
```

child

```
fork();
pid = getpid();
for (i = 1; i <= 200; i++) {
printf("This line is from pid %d, value = %d\n", pid, i);
}
return 0;
}
```

Example 1 : 2/2

```
This line is from pid 2268, value = 21
This line is from pid 2268, value = 22
This line is from pid 2268, value = 23
This line is from pid 2269, value = 3
This line is from pid 2268, value = 24
This line is from pid 2268, value = 25
This line is from pid 2269, value = 4
This line is from pid 2268, value = 26
This line is from pid 2268, value = 27
This line is from pid 2269, value = 5
This line is from pid 2268, value = 28
This line is from pid 2268, value = 29
This line is from pid 2269, value = 6
This line is from pid 2268, value = 30
This line is from pid 2269, value = 7
```

Child process 2269 and parent process 2268, both run concurrently.

2268

↓

2269

The order of these lines are determined by the CPU scheduler.

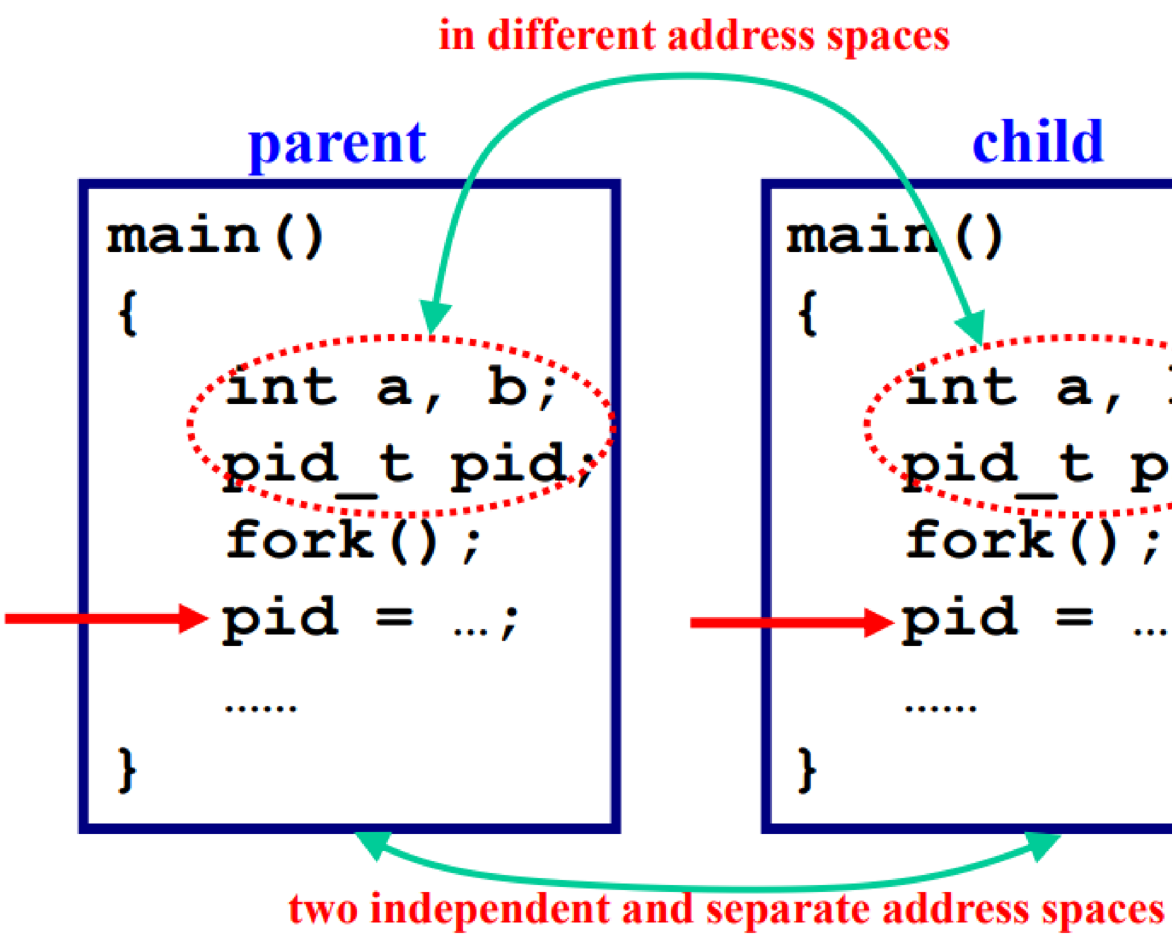
fork() Return Values

- fork() is unique (and often confusing) because it is called once but returns "twice"
- Child gets an identical (but separate) copy of the parent's address space
- Child has a different PID than the parent
- Function **getpid()** returns the process ID of the caller.
- Both processes continue/start execution after fork
- Can't predict execution order of parent and child→non deterministic

parent

```
main()
{
    int a, b;
    pid_t pid
    fork();
    pid = …;
    ......
}
```

After execution of fork()



Example 2: Separate Parent/Child Processes (1/5)

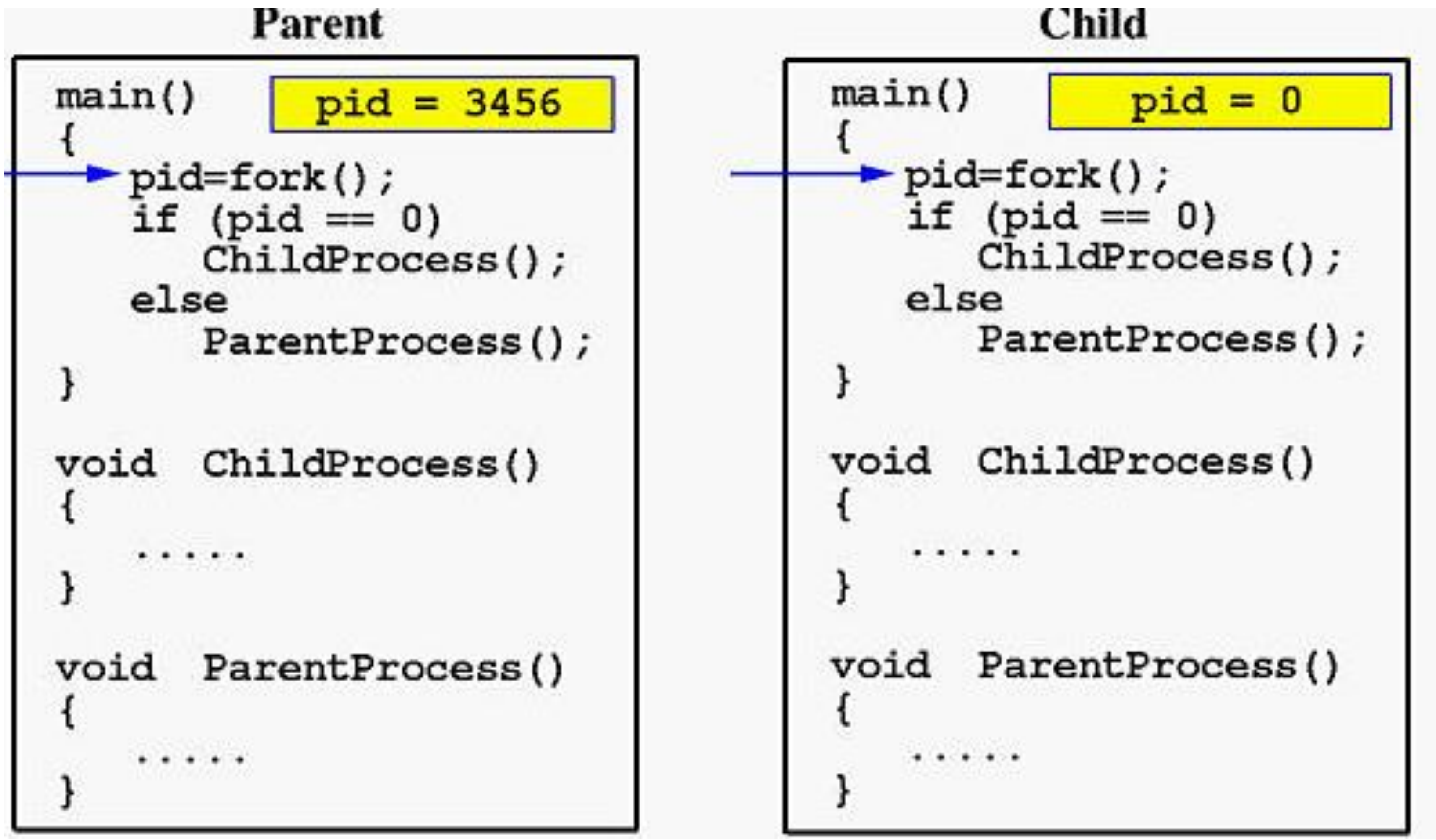
Consider a simple example, which distinguishes the parent from the child based on the value returned from fork().

```
main(void)
{
  pid_t pid;

  pid = fork();
  if (pid < 0)
    printf("Oops!");
  else if (pid == 0)
    child();
  else // pid > 0
    parent();
}
```

```
void child(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf(" Child:%d\n", i);
  printf("Child done\n");
}

void parent(void)
{
  int i;
  for (i=1; i<=10; i++)
    printf("Parent:%d\n", i);
  printf("Parent done\n");
}
```

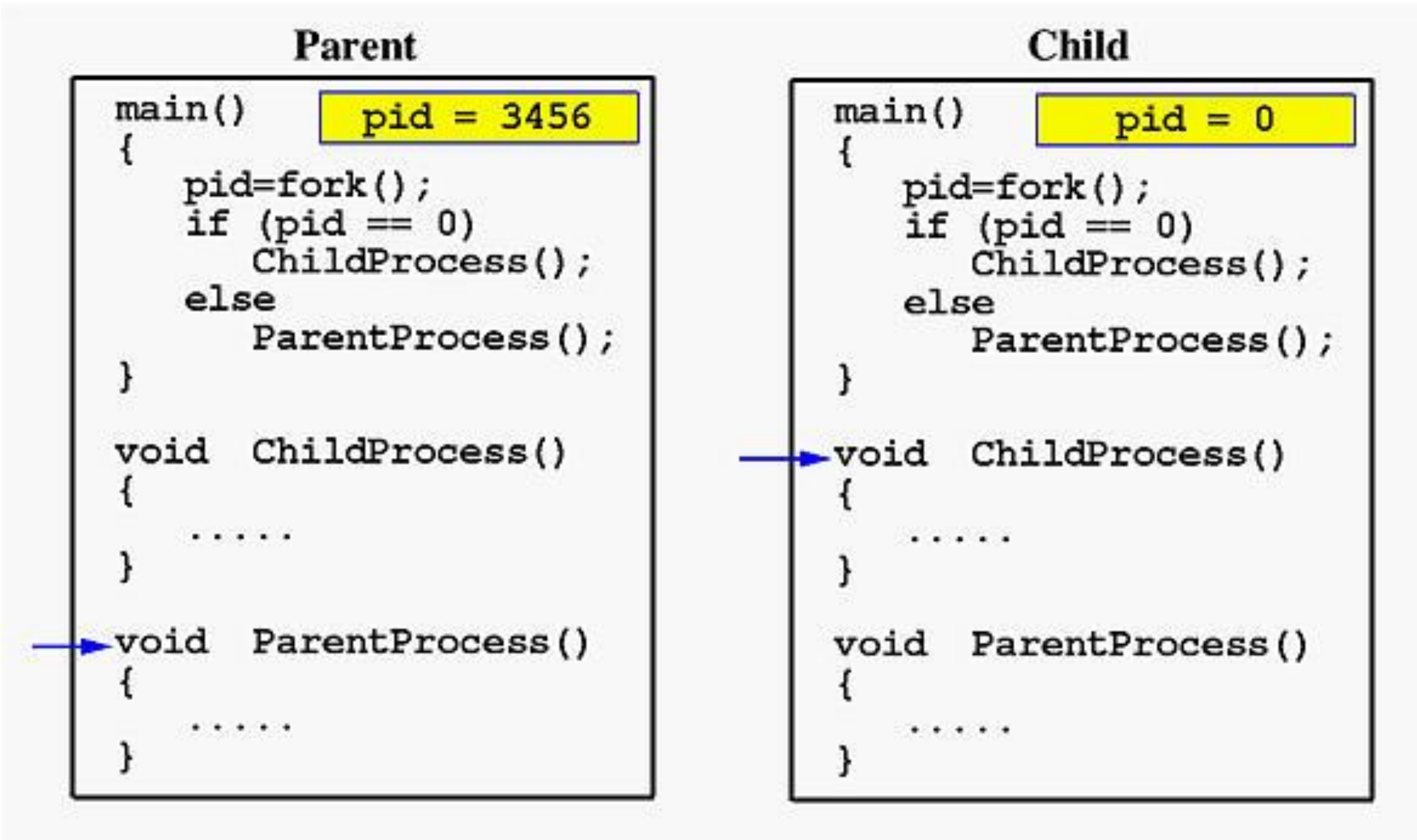
Example 2: Separate Parent/Child Processes (2/5)

Step 1: When the main program executes fork(), an identical copy of its address space, including the program and all data, is created. System call fork() returns the child process ID to the parent and returns 0 to the child process.



Example 2: Separate Parent/Child Processes (4/5)

Step 2: In the parent, since pid is non-zero, it calls function ParentProcess(). On the other hand, the child has a pid equal to zero and calls ChildProcess():



Example 2: Separate Parent/Child Processes (5/5)

```
#include <stdio.h>
#include <sys/types.h>
void  main(void) {
     pid_t  pid;
     pid = fork();
     if (pid == 0)
          ChildProcess();
     else if (pid>0)
          ParentProcess();
 }
```

Pid = 0

```
void  ChildProcess(void) {
int   i;
for (i = 1; i <= 200; i++)
printf("This line is from child, value = %d\n", i);
printf("   *** Child process is done ***\n"); }
```

Pid = 3456

```
void  ParentProcess(void) {
int   i;
for (i = 1; i <= 200; i++)
printf("This line is from parent, value = %d\n", i);
printf("*** Parent is done ***\n"); }
```

Example 2: Separate Parent/Child Processes

```
This line is from child, value = 116
This line is from child, value = 117
This line is from child, value = 118
This line is from child, value = 119
This line is from parent, value = 153
This line is from child, value = 120
This line is from parent, value = 154
This line is from child, value = 121
This line is from parent, value = 155
This line is from child, value = 122
This line is from parent, value = 156
This line is from child, value = 123
This line is from parent, value = 157
This line is from child, value = 124
This line is from parent, value = 158
This line is from child, value = 125
This line is from parent, value = 159
This line is from child, value = 126
This line is from parent, value = 160
This line is from child, value = 127
This line is from parent, value = 161
```

Both processes print lines that indicate (1) whether the line is printed by the child or by the parent process, and
(2) the value of variable i.

Class exercise 1

- Write a ‘C’ program which uses fork().
- The main program declares two variables i=10 and j=20. The program then calls fork().
    - The parent process shall print two variables and their respective values.
    - Similarly, the child shall double the values of the two variables `i`, and `j`, and then print them.

Solution

```
#include <stdlib.h>
#include <unstd.h>

void main(void)
{
pid_t pid;

int i = 10, j = 20;
if ((pid = fork()) == 0) { // child here
i = 1000; j = 2000; // child changes values
printf(" From child: i=%d, j=%d\n", i, j);

}

else { // parent here
sleep(3);
printf("From parent: i=%d, j=%d\n", i, j);

}
}
```

Orphan Process

- **An orphan process** is a process whose parent process has terminated, though it (child process) remains running itself.
- If parent terminated without waiting for the child process , the child process becomes **orphan**
- Or If a parent process terminates before its child terminates, the child process is automatically adopted by the init **process**

The **init** process periodically invokes **wait()** causing the release of orphan’s process identifier and process-table entry.

A zombie process or defunct process is a process that has completed execution but still has an entry in the process table as its parent process didn't invoke an wait() system call.

Example 3: getpid(), getppid() and Orphan Process

```
int main(void) {
  pid_t pid;
  pid = fork();

    if (pid==0){ //child Process
    printf("\From child %d: my Parent is %d\n", getpid(), getppid());
    printf("From child %d: and my Parent is now %d\n", getpid(), getppid());
    }
    else if(pid>0){ //Parent Process
    printf("From Parent %d: I spawned a child %d\n", getpid(), pid);
    printf("From Parent %d: I am Done \n", getpid());
    }
    return 0;
}
```

What happens if Parent process terminates before child process?

getpid() returns process ID

getppid() returns parent process ID

Orphan Process

```
From child 3952: My parent is: 3951
   From Parent 3951: I spawned a child 3952
   From Parent 3951: I am Done!
   From child 3952: My parent is: 1
```

Parent terminated

1

3952

While parent is still running

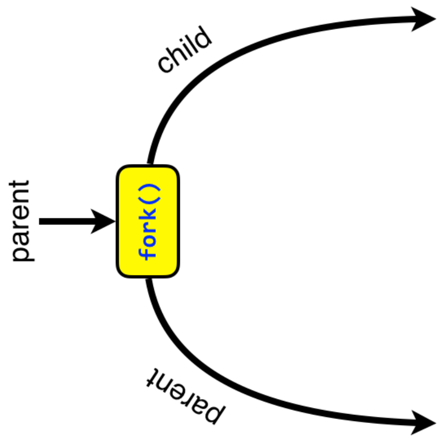
3951

3952

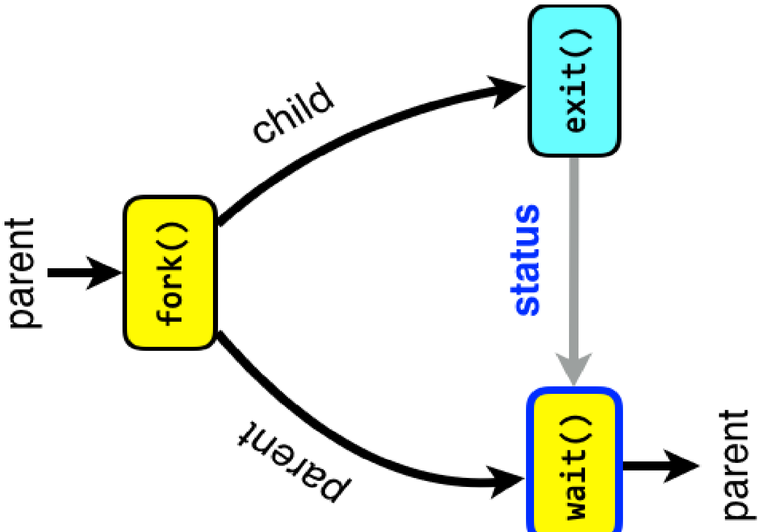
Orphan child has a new parent `init`, the Unix process that spawns all processes

The wait() system call

- So far we assumed the parent process can terminate before the child process.



- The **wait()** system call blocks the caller (parent process) until one of its child processes exits.



The wait() system call

- Synopsis of the wait() system call

```
pid_t wait(int *status);
```

  - wait() returns the pid of the terminated child process.
  - If no child process is running, `wait()` returns -1.
  - In addition, wait() takes a pointer to an integer variable **(int status)** and returns some flags that indicate the completion **`status`** of the child process are passed back with the integer pointer.

- **What** If parent process has multiple children??
  - wait will return when any of the children terminates

- `wait()` is used to synchronize parent/child execution

waitpid can be used to wait on a specific child process

How to use wait() ?

```
void main(void)
{
   pid_t pid, pid_child;
   int   status;

   if ((pid = fork()) == 0)   // child here
      child();
   else {                     // parent here
      parent();
      pid_child = wait(&status);
   }
}
```

How to use wait() ?

- Wait for an unspecified child process:

  wait(&status);

- Wait for a number, say n, of unspecified child processes:

  for (i = 0; i < n; i++)

  wait(&status);

- Wait for a specific child process whose ID is known:

  while (pid != wait(&status)) ;

```
void main(void)
{
   pid_t pid, pid_child;
   int   status;

   if ((pid = fork()) == 0)   // child here
      child();
   else {                     // parent here
      parent();
      pid_child = wait(&status);
   }
}
```

Wait: Synchronization with children

```
void main() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
      exit(0);
   } else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

What are Possible outputs?

Wait: Synchronization with children

```
void main() {
   int child_status;

   if (fork() == 0) {
      printf("HC: hello from child\n");
      exit(0);
   } else {
      printf("HP: hello from parent\n");
      wait(&child_status);
      printf("CT: child has terminated\n");
   }
   printf("Bye\n");
}
```

Two possible outputs:

HC	HP
HP	HC
CT	CT
Bye	Bye