Processes in UNIX

SDC, PPWC (CSE 3544)

Department of Computer Science & Engineering
ITER, Siksha 'O' Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Text Book(s)

- **B. M. Harwani**

  Practical C Programming

  Packt Publishing

- **Jeri R. Hanly & Elliot B. Koffman**

  Problem Solving and Program Design in C

  Seventh Edition, Pearson Education

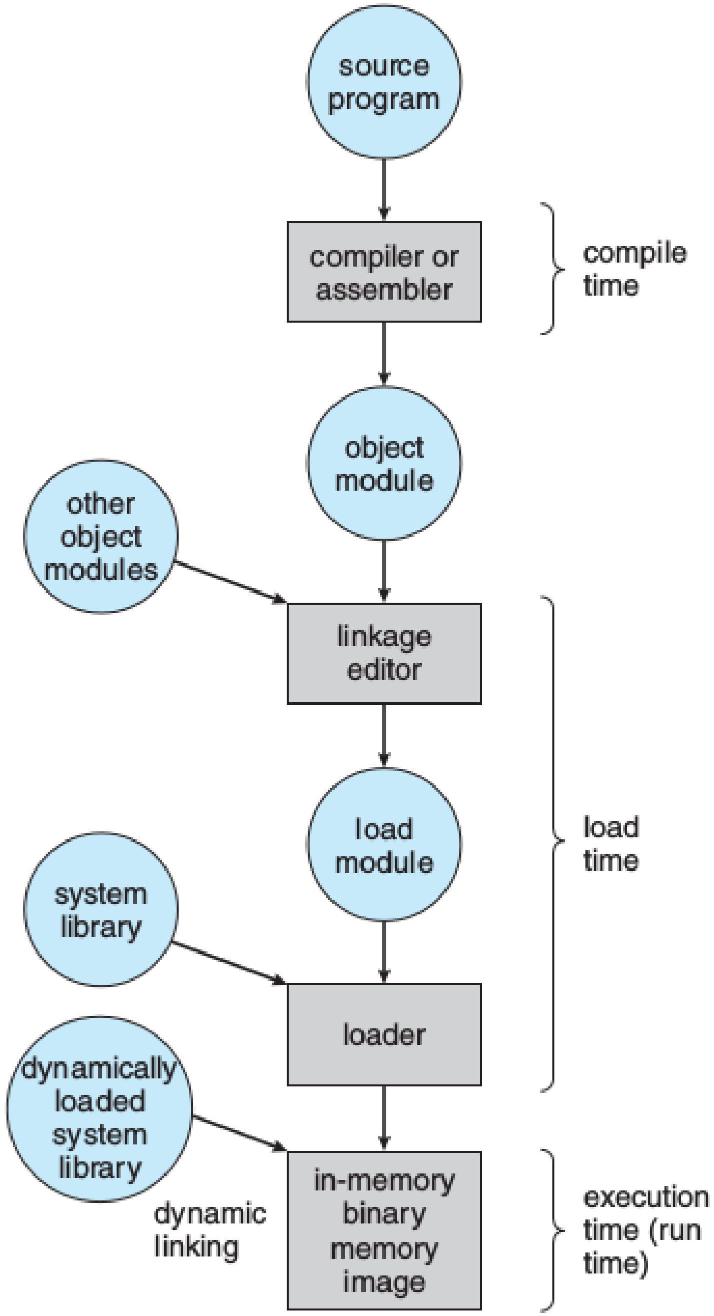
Program

- A **program** is a prepared sequence of instructions to accomplish a defined task.
- To write a C source program, a programmer creates **disk files** containing C statements that are organized into functions.
- An individual C source file may also contain variable and function declarations, type and macro definitions (e.g., typedef) and preprocessor commands (e.g., #ifdef, #include, #define).
- The source program contains exactly one `main` function.

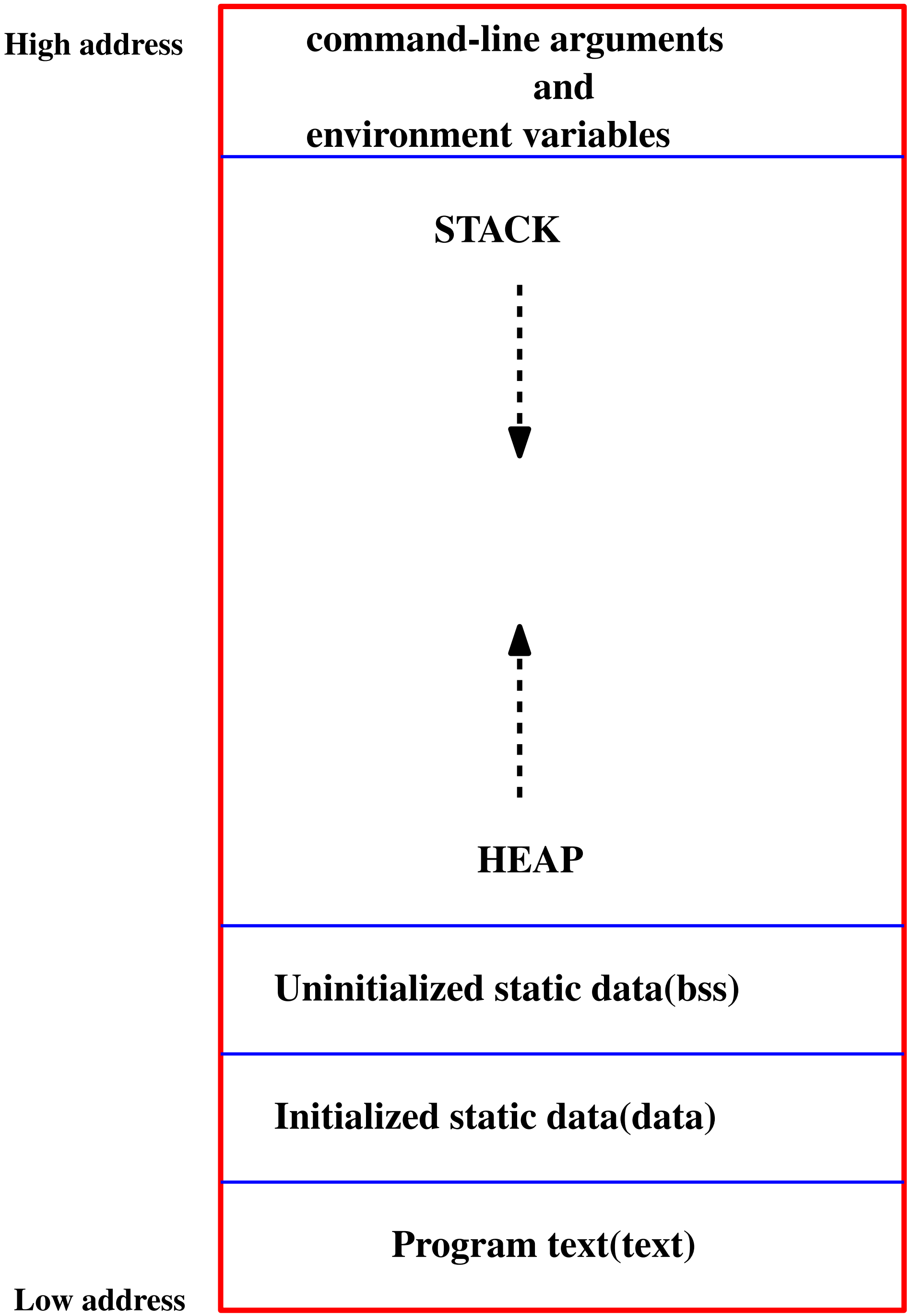
C compiler

- The C compiler translates each source file into an object file.
- The compiler then links the individual object files with the necessary libraries to produce an executable module.
- When a program is run or executed, the operating system copies the executable module into a **program image** in main memory.

Multistage Processing of a User Program



Layout of a Program Image in Main Memory



argc, **argv**, environment

activation records for function call
(return address,
parameters,
saved registers,
automatic variables)

allocation from
malloc family

command: **$ size objectfilename**

- size utility lists the section sizes and the total size for each of the object in its argument list.
- If none are specified in the filename , the file "**a.out**" will be used

Process

- A **process** is an instance of a program that is executing.

- Each instance has its own address space and execution state.

Process

- A **process** is an instance of a program that is executing.
- Each instance has its own address space and execution state.
- When does a program become a process?

When does a program become a process?

- ☞ The operating system reads the program into memory.

When does a program become a process?

- ☞ The operating system reads the program into memory.
- ☞ The allocation of memory for the program image is not enough to make the program a process.

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.
- The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.
- The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.
- The operating system also manages the memory occupied by the processes and the memory available for allocation.

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.
- The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.
- The operating system also manages the memory occupied by the processes and the memory available for allocation.
- **When the operating system has added the appropriate information in the kernel data structures and has allocated the necessary resources to run the program code, the program has become a process.**

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.
- The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.
- The operating system also manages the memory occupied by the processes and the memory available for allocation.
- **When the operating system has added the appropriate information in the kernel data structures and has allocated the necessary resources to run the program code, the program has become a process.**
- A process has an address space (memory it can access) and at least one flow of control called a thread.

When does a program become a process?

- The operating system reads the program into memory.
- The allocation of memory for the program image is not enough to make the program a process.
- The process must have an ID (the *process ID*) so that the operating system can distinguish among individual processes. The *process state* indicates the execution status of an individual process.
- The operating system keeps track of the process IDs and corresponding process states and uses the information to allocate and manage resources for the system.
- The operating system also manages the memory occupied by the processes and the memory available for allocation.
- **When the operating system has added the appropriate information in the kernel data structures and has allocated the necessary resources to run the program code, the program has become a process.**
- A process has an address space (memory it can access) and at least one flow of control called a thread.
- The variables of a process can either remain in existence for the life of the process (static storage) or be automatically allocated when execution enters a block and deallocated when execution leaves the block (automatic storage).

Threads and Thread of Execution

☞ When a program executes, the value of the process program counter determines which process instruction is executed next.

Threads and Thread of Execution

- When a program executes, the value of the process program counter determines which process instruction is executed next.
- The resulting stream of instructions, called a thread of execution

Threads and Thread of Execution

- When a program executes, the value of the process program counter determines which process instruction is executed next.
- The resulting stream of instructions, called a thread of execution
- A thread of execution can be represented by the sequence of instruction addresses assigned to the program counter during the execution of the program's code.

Example-1

Process 1 executes statements 245, 246 and 247 in a loop. Its thread of execution can be represented as 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1 . . . , where the subscripts identify the thread of execution as belonging to process 1.

Threads and Thread of Execution

- When a program executes, the value of the process program counter determines which process instruction is executed next.
- The resulting stream of instructions, called a thread of execution
- A thread of execution can be represented by the sequence of instruction addresses assigned to the program counter during the execution of the program's code.

Example-1

Process 1 executes statements 245, 246 and 247 in a loop. Its thread of execution can be represented as 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1, 247_1 . . . , where the subscripts identify the thread of execution as belonging to process 1.

Example-2

Process 1 executes its statements 245, 246 and 247 in a loop as in Example 1, and process 2 executes its statements 10, 11, 12 . . . . The CPU executes instructions in the order 245_1, 246_1, 247_1, 245_1, 246_1, 247_1, 245_1, 246_1 , [context-switch instructions], 10_2 , 11_2 , 12_2 , 13_2 , [context-switch instructions], 247_1, 245_1, 246_1, 247_1 . . . . Context switches occur between 246_1 and 10_2 and between 13_2 and 247_1 . The processor sees the threads of execution interleaved, whereas the individual processes see uninterrupted sequences.

Process Identification

- UNIX identifies processes by a unique integral value called the process ID.
- Each process also has a parent process ID, which is initially the process ID of the process that created it.
- If this parent process terminates, the process is adopted by a system process so that the parent process ID always identifies a valid process.

getpid and getppid Function

- The **getpid** function returns the process ID .
- The **getppid** function returns the parent process ID.
- The **pid_t** is an unsigned integer type that represents a process ID.

```
#include <unistd.h>

pid_t getpid(void);

pid_t getppid(void);
```

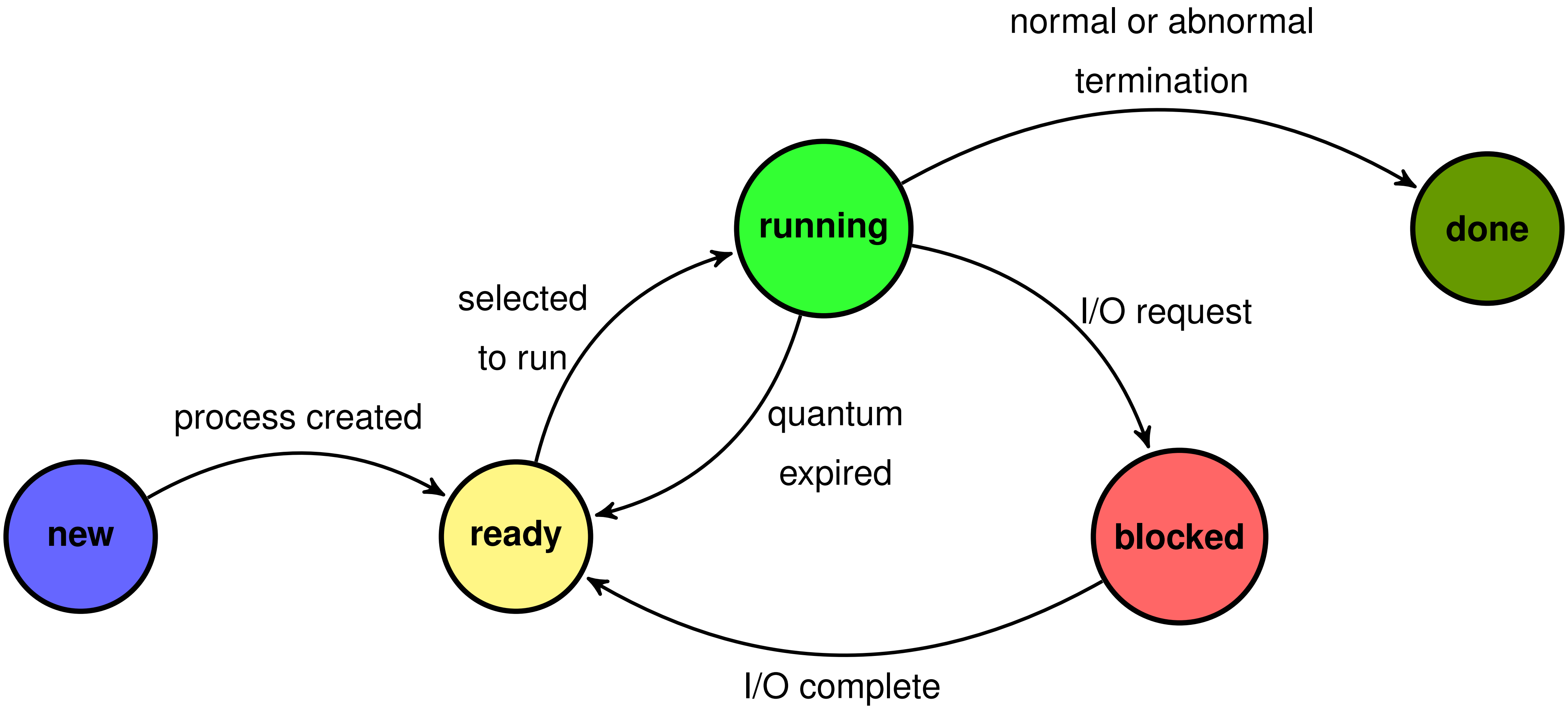
Neither the **getpid** nor the **getppid** functions can return an error.

Example

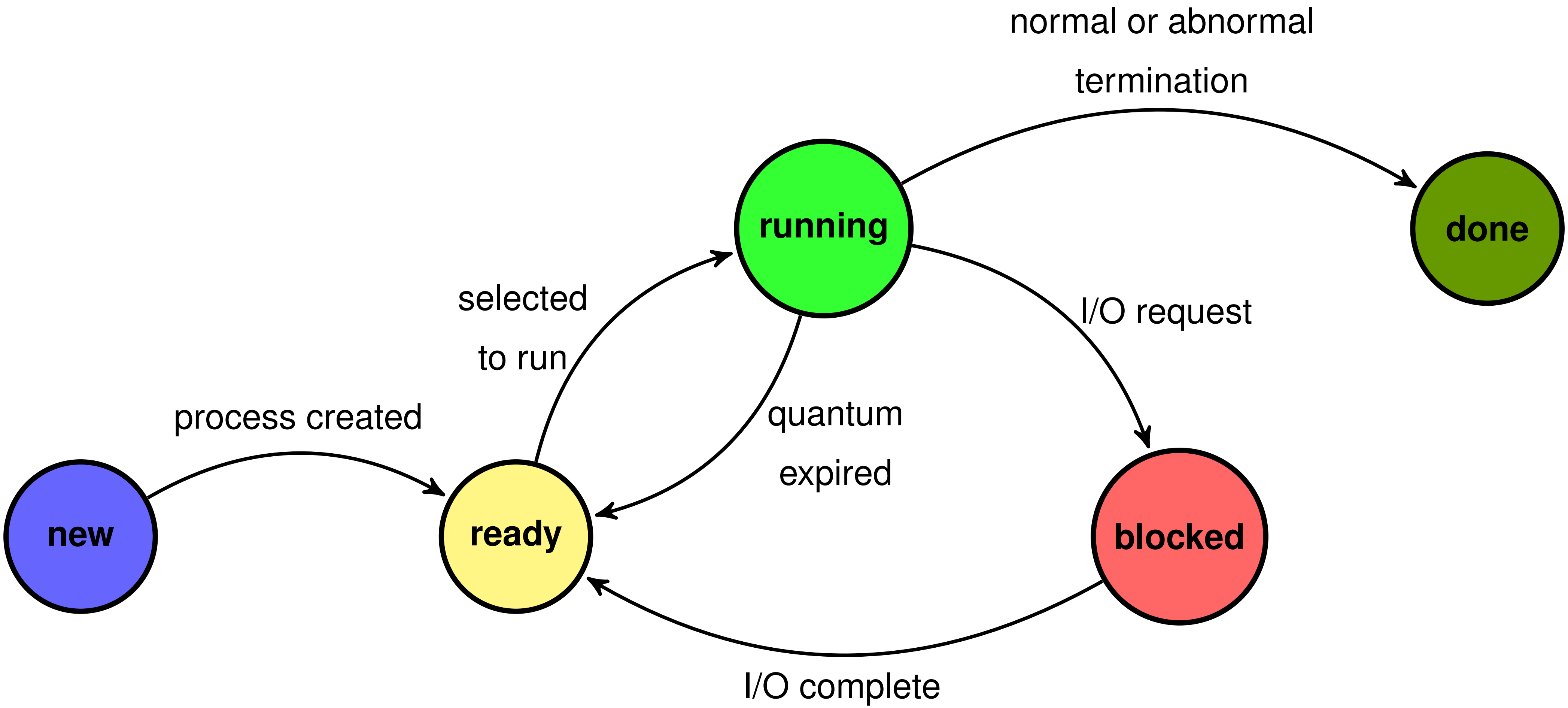
```
#include <stdio.h>
#include <unistd.h>
int main (void) {
printf("I am process %ld\n", (long)getpid());
printf("My parent is %ld\n", (long)getppid());
return 0;
}
```

Process State

Process State



Process State



state	meaning
new	being created
ready	waiting to be assigned to a processor
running	instructions are being executed
blocked	waiting for an event such as I/O
done	finished

`ps` Utility

`ps` displays information about a selection of the active processes. If you want a repetitive update of the selection and the displayed information, use **top** instead.

- ✍ **`$ ps`** displays information about processes associated with the user.
- ✍ The **`& ps -a`** option displays information for processes associated with all the terminals.
- ✍ The **`& ps -A`** option displays information for all processes.
- ✍ **`& ps -la`** long format
- ✍ Try more ......

UNIX Process Creation and `fork`

- A process can create a new process by calling **`fork`**.
- The calling process becomes the **`parent`**, and the created process is called the **`child`**.
- The **`fork`** function copies the parent's memory image so that the new process receives a copy of the address space of the parent.
- Both processes continue at the instruction after the **`fork`** statement (executing in their respective memory images).
- **`fork()`** prototype

```
#include <unistd.h>

pid_t fork(void);
```

- **`fork()`** function returns

```
(1) returns 0 to the child
(2) returns the child's process ID to the parent.
(3) When fork fails, it returns -1 and
    sets the errno.
```

UNIX Process Creation and `fork`

- A process can create a new process by calling **`fork`**.
- The calling process becomes the **`parent`**, and the created process is called the **`child`**.
- The **`fork`** function copies the parent's memory image so that the new process receives a copy of the address space of the parent.
- Both processes continue at the instruction after the **`fork`** statement (executing in their respective memory images).
- **`fork()`** prototype

```
#include <unistd.h>

pid_t fork(void);
```

- **`fork()`** function returns

```
(1) returns 0 to the child
(2) returns the child's process ID to the parent.
(3) When fork fails, it returns -1 and
    sets the errno.
```

Note:: The fork function return value is the critical characteristic that allows the parent and the child to distinguish themselves and to execute different code.

More About `fork`

- If the system does not have the necessary resources to create the child or if limits on the number of processes would be exceeded, **`fork`** sets `errno` to `EAGAIN`.

- In case of a failure, the **`fork`** doesnot create a child.

Find the Output

```c
#include <stdio.h>
#include <unistd.h>
int main(void) {
    int x;
    x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x
              is %d\n", (long)getpid(), x);
    return 0;
}
```

Find the Output

```
#include <stdio.h>
#include <unistd.h>
int main(void) {
    int x;
    x = 0;
    fork();
    x = 1;
    printf("I am process %ld and my x
              is %d\n", (long)getpid(), x);
    return 0;
}
```

In the above program, both parent and child execute the x = 1 assignment statement after returning from **fork**.

Testing the return value of `fork`

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
   pid_t childpid;
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                          /* child code */
      printf("I am child %ld\n", (long)getpid());
   else                                        /* parent code */
      printf("I am parent %ld\n", (long)getpid());
   return 0;
}
```

Testing the return value of `fork`

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
   pid_t childpid;
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                  /* child code */
      printf("I am child %ld\n", (long)getpid());
   else                                /* parent code */
      printf("I am parent %ld\n", (long)getpid());
   return 0;
}
```

After **`fork`** in the above program, the parent and child output their respective process IDs.

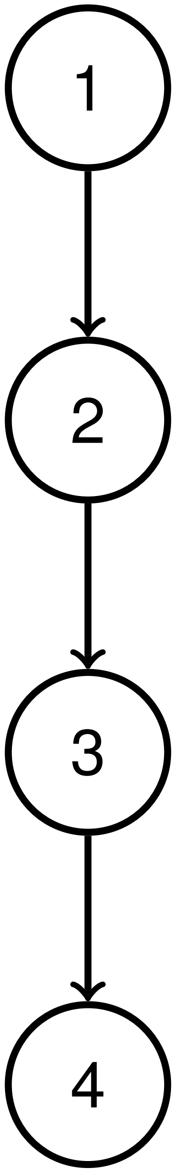
Find the Output

```
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
   pid_t childpid;
   pid_t mypid;
   mypid = getpid();
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                 /* child code */
      printf("I am child %ld, ID = %ld\n", (long)
          getpid(), (long)mypid);
   else                               /* parent code */
      printf("I am parent %ld, ID = %ld\n", (long)
          getpid(), (long)mypid);
   return 0;
}
```

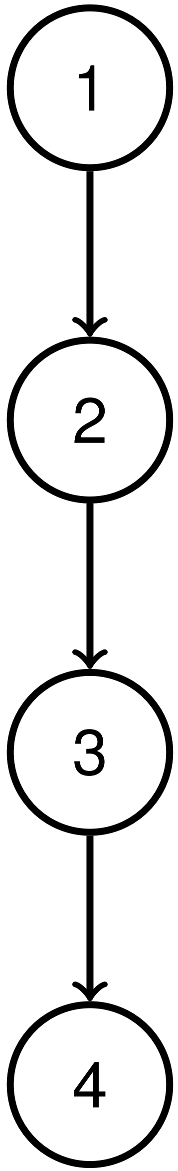
Find the Output

```c
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
int main(void) {
   pid_t childpid;
   pid_t mypid;
   mypid = getpid();
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0)                         /* child code */
      printf("I am child %ld, ID = %ld\n", (long)
          getpid(), (long)mypid);
   else                                       /* parent code */
      printf("I am parent %ld, ID = %ld\n", (long)
          getpid(), (long)mypid);
   return 0;
}
```

Chain of n=4 Processes



Chain of n=4 Processes



- ☞ A graph representing the chain of processes, when n is 4.
- ☞ Each circle represents a process labeled by its value of **i** when it leaves the loop as per implementation given in next slide.
- ☞ The edges represent the **is-a-parent** relationship.
- ☞ A $\longrightarrow$ B means process A is the parent of process B.

Creating a Chain of n Processes

Create a chain of n processes by calling fork in a loop.

Creating a Chain of n Processes

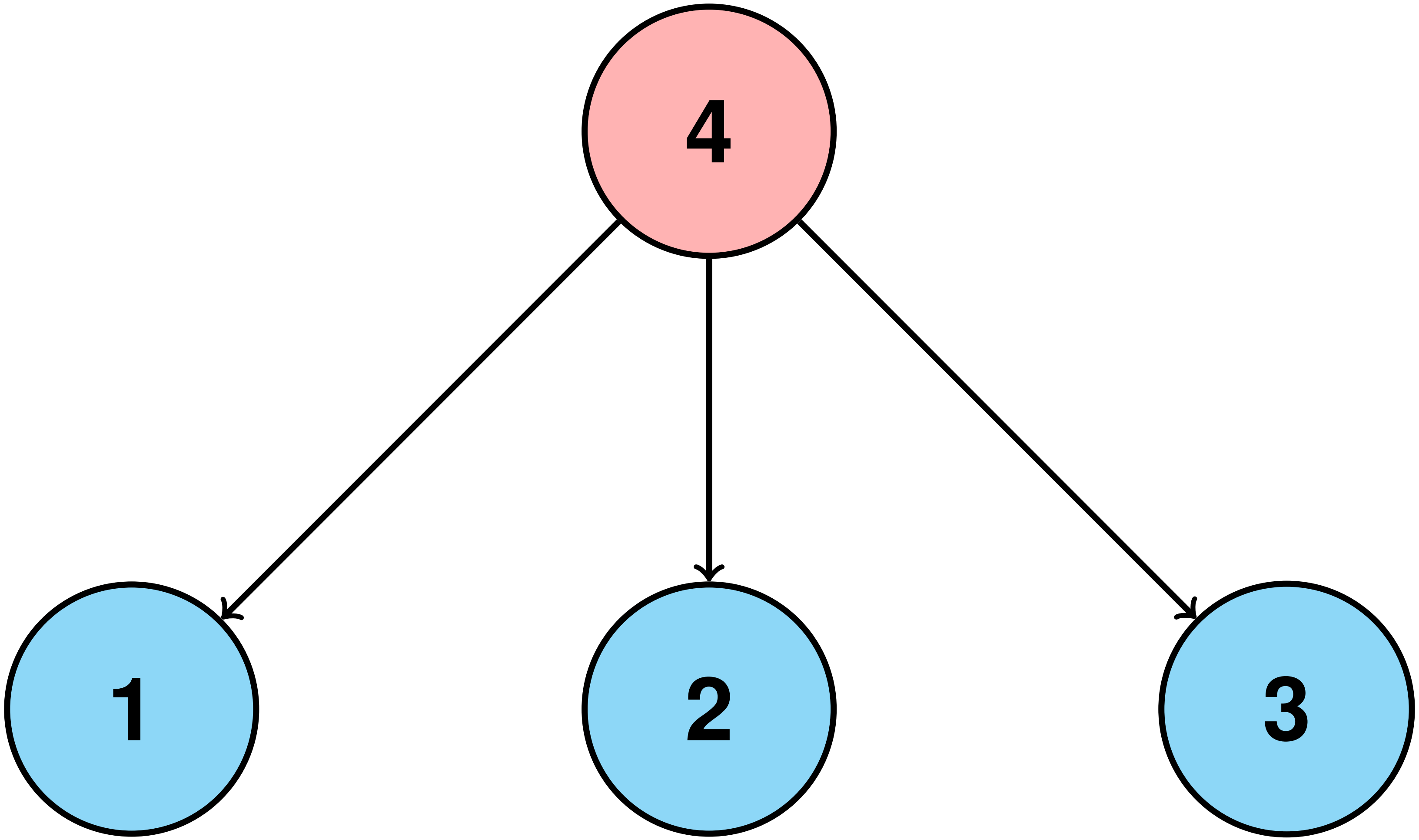
Create a chain of n processes by calling fork in a loop.

```
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;
   if (argc != 2){/* check for valid number of
       command-line arguments */
      fprintf(stderr, "Usage: %s processes\n", argv
          [0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
      if (childpid=fork())
         break;
   fprintf(stderr, "i:%d process ID:%ld parent ID:%
       ld child ID:%ld\n", i, (long)getpid(), (long)
       getppid(), (long)childpid);
   return 0;
}
```

Briefing of the Previous Code

- Creates a chain of n processes by calling **`fork`** in a loop.
- On each iteration of the loop, the parent process has a nonzero childpid and hence breaks out of the loop.
- The child process has a zero value of childpid and becomes a parent in the next loop iteration.
- In case of an error, **`fork`** returns -1 and the calling process breaks out of the loop.

A Fan of n=4 Processes



Creating a Fan of n Processes

Creates a fan of n processes by calling **`fork`** in a loop.

Creating a Fan of n Processes

Creates a fan of n processes by calling **fork** in a loop.

```
int main (int argc, char *argv[]) {
   pid_t childpid = 0;
   int i, n;
   if (argc != 2){
   /* check for valid number of command-line
       arguments */
      fprintf(stderr, "Usage: %s processes\n", argv
          [0]);
      return 1;
   }
   n = atoi(argv[1]);
   for (i = 1; i < n; i++)
     if ((childpid = fork()) <= 0)
         break;
   fprintf(stderr, "i:%d process ID:%ld parent ID:%
       ld child ID:%ld\n",i, (long)getpid(), (long)
       getppid(), (long)childpid);
   return 0;
}
```

Workout Exercise

Explain what happens when you replace the test

```
(childpid = fork()) <= 0
```

of the previous slide program with

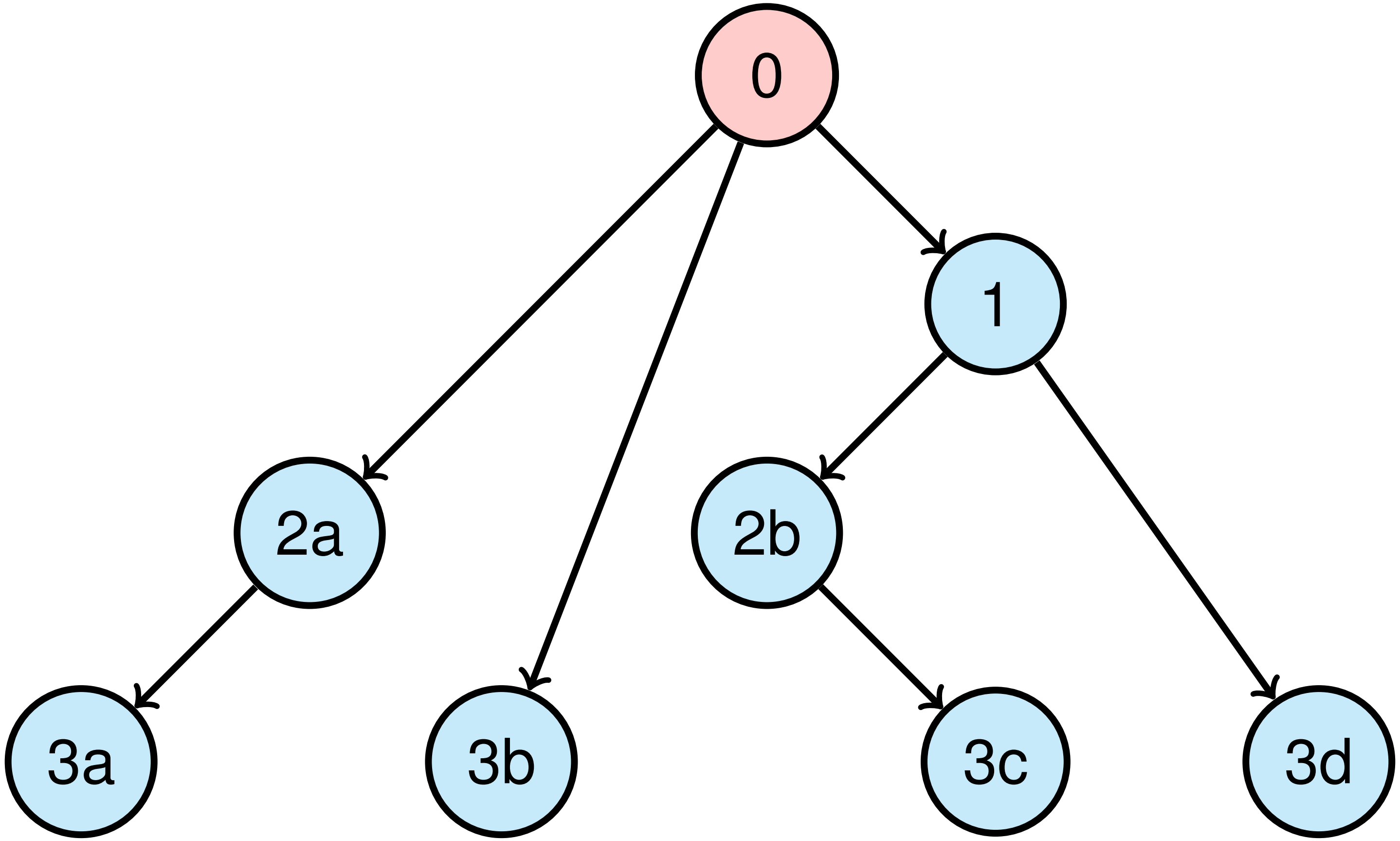
```
(childpid = fork()) == -1
```

Draw a suitable diagram labelling the circles with the actual process.

Solution to Previous Exercise

Tree of processes produced by the modification of the test condition of the Program

```
(childpid = fork()) == -1
```



Parent Vs Child in `fork`

`fork` - create a child process

Description: fork() creates a new process by duplicating the calling process. The new process, referred to as the **child**, is an exact duplicate of the calling process, referred to as the **parent**, **except for the following points:**

- ✍ The child has its own unique process ID, and this PID does not match the ID of any existing process group.
- ✍ The child's parent process ID is the same as the parent's process ID.
- ✍ The child does not inherit its parent's memory locks.
- ✍ Process resource utilizations and CPU time counters are reset to zero in the child.
- ✍ The child's set of pending signals is initially empty.
- ✍ The child does not inherit semaphore adjustments from its parent.
- ✍ The child does not inherit record locks from its parent.
- ✍ The child does not inherit timers from its parent.
- ✍ The child does not inherit outstanding asynchronous I/O operations from its parent, nor does it inherit any asynchronous I/O contexts from its parent.

Parent Vs Child in `fork` Contd...

- The child does not inherit directory change notifications from its parent.
- **The child does not receive a signal when its parent terminates.**
- The termination signal of the child is always **`SIGCHLD`**.
- The port access permission bits set by **ioperm** (ioperm - set port input/output permissions) are not inherited by the child
- **The child process is created with a single thread**-the one that called fork(). The entire virtual address space of the parent is replicated in the child, including the states of mutexes, condition variables, and other pthreads objects.
- **The child inherits copies of the parent's set of open file descriptors.** Each file descriptor in the child refers to the same open file description as the corresponding file descriptor in the parent.
- The child inherits copies of the parent's set of open message queue descriptors.
- The child inherits copies of the parent's set of open directory streams.

What Does the Parent Do While the Child Executes

It can do two things:

- ✍ Wait to gather the child's exit status.
- ✍ Continue execution without waiting for the child(and pick up the exit status later, if at all)

Two system calls invoked in waiting **`wait()`** and **`waitpid()`**

Process Termination

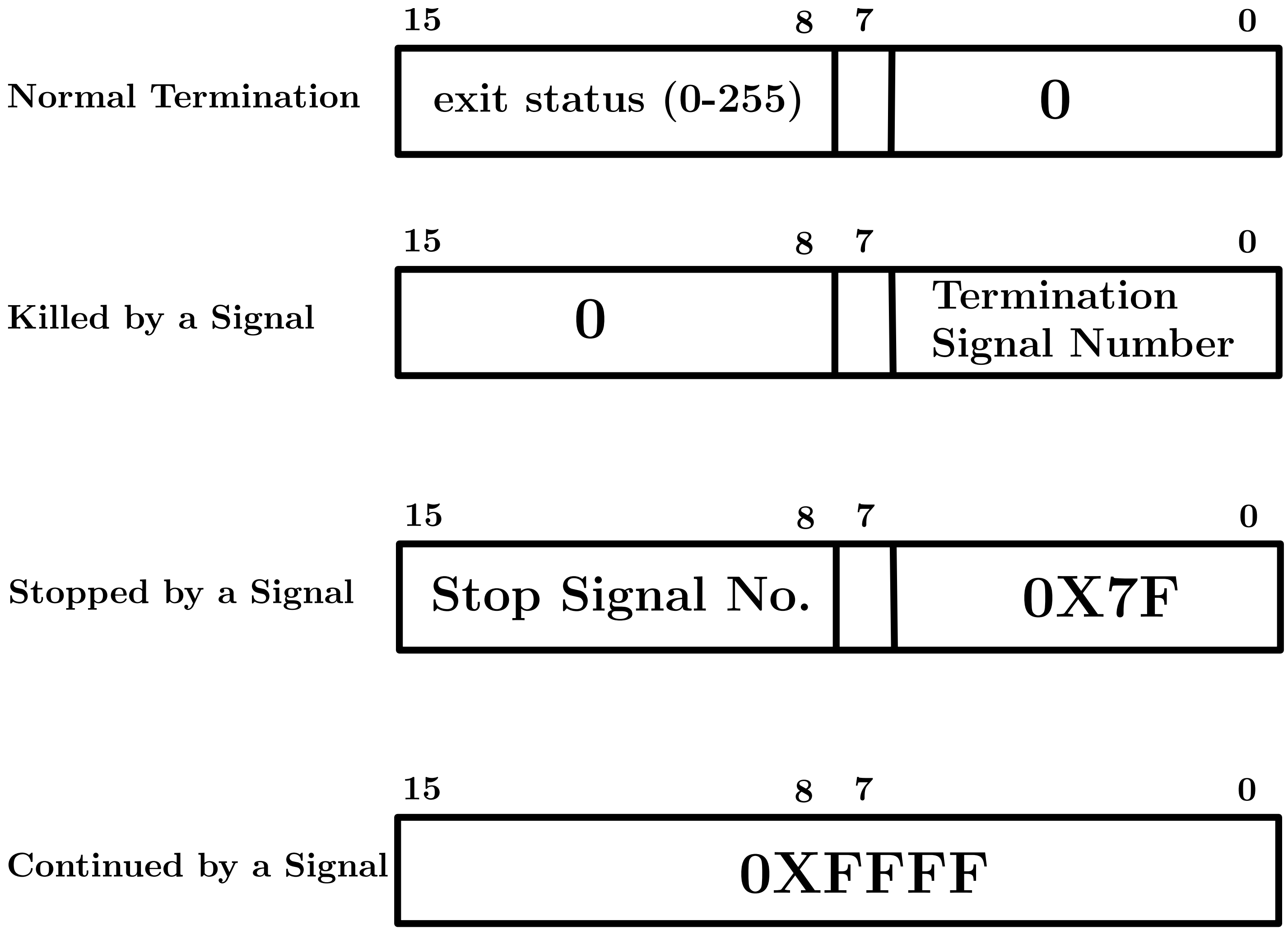
- When a process terminates, the operating system deallocates the process resources, updates the appropriate statistics and notifies other processes of the demise.
- The termination can either be **normal** or **abnormal**.
- The activities performed during process termination:
  - Canceling pending timers and signals,
  - Releasing virtual memory resources,
  - Releasing other process-held system resources such as locks, and closing files that are open.
- The operating system records the process status and resource usage, notifying the parent in response to a wait function.

Process Termination

- In UNIX, a process does not completely release its resources after termination until the parent waits for it.

- If its parent is not waiting when the process terminates, the process becomes a **zombie**.
- A zombie is an inactive process whose resources are deleted later when its parent **waits** for it.
- A **normal termination** occurs under the following conditions;
  - `return` from `main`
  - Implicit return from main (the main function falls off the end)
  - Call to `exit`, `_Exit` or `_exit`

Process Termination



7^{th} bit in killed and stopped by a signal is for CORE file

(1) 7^{th} bit=1; **Core file generated**

(2) 7^{th} bit=0; **Core file not generated**

ZOMBIE and ORPHAN

Two things can be happened, if the the parent doesnot wait for the child to die.

- ✍ The child dies while the parent is still alive
- ✍ The parent dies while the child is still alive

ZOMBIE

- ✍ If the child dies first, the kernel empties the process address space but retains the process table entry. The child is said to be in a **zombie** state.
- ✍ The zombie is actually not a process at all, so canot be killed.
- ✍ The only reason for a child to memain in the zombie state is the hope that the parent may eventually call **`wait`** or **`waitpid`** to pickup the exit status and clear the process table slot.
- ✍ Type **`man ps`** to read process state information and locate the state zombie

Example: ZOMBIE

```
int main()
{
  pid_t childpid;
  childpid=fork();
  if(childpid==-1){
     printf("fork error\n");
     return 1;
  }
  else if (childpid==0){
     printf(" I am child my process ID=%ld\n",(long)getpid());
     exit(0);
  }
  else{
     printf("I am parent My PID=%ld\n",(long)getpid());
     sleep(100);
     wait(NULL);
     exit(0);
  }
}
```

How to Identify ZOMBIE ?

- ✍ Open two terminal windows. Run the previous slide code in one terminal. On the other terminal run the command
  `(ps -la | grep CMD) ; (ps -la | grep a.out)` ↵
- ✍ Look into the line that contains **`defaunct`**. It is the Zombie with process state code **Z**
- ✍ After the program terminate, again type the command **`ps -la`**. Is the ZOMBIE exit or not? If not process terminate.
- ✍ For more **`man ps`** to read process state information under the heading **PROCESS STATE CODES**

ORPHAN

- ✍ When the parent dies first, the child becomes an **orphan** since the parent is not just there to pickup the child's exit status.
- ✍ The kernel clears the process table slot of the parent, but before doing so, it checks whether there are any process spawned by the parent that are still alive.
- ✍ When it finds one , it makes **`init`**/ **`systemd`** its parent by changing **PPID** field of the child.

Example: ORPHAN

```c
int main()
{
  pid_t childpid;
  childpid=fork();
  if(childpid==-1){
    printf("fork error\n");
    return 1;
  }
  else if (childpid==0){
    printf("Child:PID=%ld---PPID=%ld\n",(long)getpid(),(long)getppid());
    sleep(100);
    printf("Child:PID=%ld---PPID=%ld\n",(long)getpid(),(long)getppid());
  }
  else{
    printf("Parent:PID=%ld--PPID=%ld\n",(long)getpid(),(long)getppid());
    exit(0);
  }
}
```

How to Identify ORPHAN ?

- Open two terminal windows. Run the previous slide code in one terminal. On the other terminal run the command **`(ps -le | grep CMD) ; (ps -le | grep <ChildPID>)`**
- Observe the process id (PID) and parent id (PPID) of the child and parent
- For more **`man ps`** to read process state information

Prototype of `exit`, `_Exit` and `_exit`

```
#include <stdlib.h>

void exit(int status);

void _Exit(int status);
                                      ISO C Version
```

```
#include <unistd.h>

void _exit(int status);
                                      POSIX Version
```

`atexit()` Function

```
#include <stdlib.h>

int atexit(void (*function)(void));
                                                  ISO C Version
Return value:
        (1) the value 0 if successful;
        (2) otherwise it returns a nonzero value.
```

- ✍ The **`atexit()`** function registers the given function to be called at normal process termination, either via **`exit()`** or via return from the program's **`main()`**
- ✍ Functions so registered are called in the reverse order of their registration; no arguments are passed.
- ✍ The same function may be registered multiple times: it is called once for each registration.

Example: `atexit()` Function

```c
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>
void display()
{
  printf("function Exit handler\n");
}
int main(void) {
  int i;
  if (atexit(display)) {
    fprintf(stderr, "Failed to install display exit handler\n");
    return 1;
}
/* rest of main program goes here */
  for(i=0;i<5;i++){
    sleep(1);
    printf("%d\n",i);
  }
  return 0;
}
```

wait and **waitpid** System Calls

- **wait** and **waitpid:** wait for process to change state
- These system calls are used to wait for state changes in a child of the calling process, and obtain information about the child whose state has changed.
- A state change is considered to be:
  - the child terminated: Normal termination (success/unsuccess);
  - the child was killed by a signal
  - the child was stopped by a signal; or
  - the child was resumed (i.e. continued) by a signal.
- In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; If a wait is not performed, then the terminated child remains in a "zombie" state
- If a child has already changed state, then these calls return immediately. Otherwise they block until either a child changes state or a signal handler interrupts the call.

The `wait` Function

- When a process creates a child, both parent and child proceed with execution from the point of the fork.
- The parent can execute the system calls **`wait`** or **`waitpid`** to block until the child finishes.
- The **`wait`** function causes the caller to suspend execution until a child's status becomes available or until the caller receives a signal.
- A process status most commonly becomes available after termination, but it can also be available after the process has been stopped.
- **`wait`** synopsis

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

pid_t wait(int *stat_loc);
```

- If **`wait`** returns because the status of a child is reported, these functions return the **process ID of that child**.
- If an error occurs, these functions return -1 and set `errno`.

Wait Example-1

```
#include<stdio.h>
#include<unistd.h>
#include<sys/wait.h>
#include<sys/types.h>
int main(void)
{
   pid_t childpid;
   childpid=fork();
   if (childpid==0) /*child*/
      printf("Inside child \n");
   else{
      wait(NULL);
      printf("In parent\n");
   }
   return 0;
}
```

Wait Example-2

```
int main(void)
{
   pid_t firstCh, secondCh;
   pid_t returnR;
   int sum;
   firstCh=fork();
   if(firstCh>0){
      printf("Parent Section\n");
      returnR= wait(NULL);
      printf("After the child process termination
          wait return value=%ld\n", (long)returnR);
   }
   if(firstCh==0){
      printf("The child process id=%ld\n", (long)
          getpid());
      sum=20+30;
      printf("sum=%d\n",sum);
      printf("Child Completes\n");
   }
   return 0;
}
```

Testing Child's Return Status

- The **stat_loc** argument of **wait** or **waitpid** is a pointer to an integer variable. If it is not NULL, these functions store the return status of the child in this location.
- The child returns its status by calling **exit**, **_exit**, **_Exit** or **return** from main. A zero return value indicates EXIT_SUCCESS; any other value indicates EXIT_FAILURE.
- **The parent can only access the 8 least significant bits of the child's return status.**
- POSIX specifies **six** macros for testing the child's return status. Each takes the status value returned by a child to **wait** or **waitpid** as a parameter.

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

WIFEXITED(int stat_val)----------------WEXITSTATUS(int stat_val)
WIFSIGNALED(int stat_val)--------------WTERMSIG(int stat_val)
WIFSTOPPED(int stat_val)---------------WSTOPSIG(int stat_val)
WIFCONTINUED(int stat_val)-------------SIGCONT
```

Macro Evaluation

- The six macros are designed to be used in pairs.
- The **WIFEXITED** evaluates to a nonzero value when the child terminates normally.
- If **WIFEXITED** evaluates to a nonzero value, then **WEXITSTATUS** evaluates to the low-order 8 bits returned by the child through **_exit()**, **exit()** or **return** from **main**.
- The **WIFSIGNALED** evaluates to a nonzero value when the child terminates because of an uncaught signal.
- If **WIFSIGNALED** evaluates to a nonzero value, then **WTERMSIG** evaluates to the number of the signal that caused the termination.
- The **WIFSTOPPED** evaluates to a nonzero value if a child is currently stopped.
- If **WIFSTOPPED** evaluates to a nonzero value, then **WSTOPSIG** evaluates to the number of the signal that caused the child process to stop.

Macros to Evaluate the Termination Status Returned by `wait` and `waitpid`

Macro	Description
`WIFEXITED(status)`	Returns True if the child terminated normally. In this case, execute **`WEXITSTATUS(status)`** to fetch the low-order 8 bits of the argument that the child passed to **`exit`**, **`_exit`**, or **`_Exit`**.
`WIFSIGNALED(status)`	Returns True if the child process was terminated by a signal. In this case, execute **`WTERMSIG(status)`** to fetch the signal number that caused the termination. Additionally, some implementations define the macro **`WCOREDUMP(status)`** that returns true if a core file of the terminated process was generated.
`WIFSTOPPED`	Returns True if the child process was stopped by delivery of a signal. In this case, execute **`WSTOPSIG(status)`** to fetch the signal number that caused the child to stop.
`WIFCONTINUED(status)`	Returns True if the child process was resumed by delivery of **`SIGCONT`**.

Macro Evaluation

Testing		Getting Status
	Normal Termination	
WIFEXITED(status)	15 ... 8 | 7 ... 0 — exit status (0-255) | | 0	**WEXITSTATUS(status)**
	Killed by a Signal	
WIFSIGNALED(status)	15 ... 8 | 7 ... 0 — 0 | | Termination Signal Number	**WTERMSIG(status)**
	Stopped by a Signal	
WIFSTOPPED(status)	15 ... 8 | 7 ... 0 — Stop Signal No. | | 0X7F	**WSTOPSIG(status)**
	Continued by a Signal	
WIFCONTINUED(status)	15 ... 8 7 ... 0 — 0XFFFF	Resumed by **SIGCONT**

The parent can only access the 8 least significant bits of the child's return status.

A Sample Code: Macro Evaluation

```
int main(void){
   pid_t childpid,pid;
   int status;
   pid=fork();
   if(pid==0){
      printf("Child Part Executed!!!!\n");
      exit(0);
   }
  else{
      childpid=wait(&status);
     if (childpid == -1)
         perror("Failed to wait for child\n");
     else if (WIFEXITED(status) && !WEXITSTATUS(status))
         printf("Child %ld terminated normally\n", (long)childpid);
     else if (WIFEXITED(status))
         printf("Child %ld terminated with return status %d\n",(long)childpid
              , WEXITSTATUS(status));
     else if (WIFSIGNALED(status))
         printf("Child %ld terminated due to uncaught signal %d\n",(long)
              childpid, WTERMSIG(status));
     else if (WIFSTOPPED(status))
         printf("Child %ld stopped due to signal %d\n",(long)childpid,
              WSTOPSIG(status));
 }
 return 0;}
```

Limitations of `wait()` System Call

- ✍ It is not possible for the parent to retrieve the signal number during which the child process has stopped the execution (`SIGSTOP(19)`, `SIGTSTP(20)`).
- ✍ Parent won't be able to get the notification when a stopped child was resumed by the delivery of a signal (`SIGCONT(18)`, `SIGCHLD(17)`).
- ✍ Parent can only wait for first child that terminates. It is not possible to wait for a particular child.
- ✍ It is not possible for a non blocking wait so that if no child has yet terminated, parent get an indication of this fact.

The `waitpid` Function

- The waitpid function takes **three** parameters:
  - a pid,
  - a pointer to a location for returning the status
  - a flag specifying options.

- **`waitpid`** synopsis

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- If **`waitpid`** returns because the status of a child is reported, these functions return the **process ID of that child**.

- If an error occurs, these functions return -1 and set `errno`.

First Parameter of `waitpid` Function

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- If **`pid`** $= -1$, **`waitpid`** waits for any child. In this respect, **`waitpid`** is equivalent to **`wait`**.
- If **`pid`** > 0, **`waitpid`** waits for the child whose process ID equals **`pid`**.
- If **`pid`** $= 0$, **`waitpid`** waits for any child whose process group ID equals that of the calling process.
- If **`pid`** < -1, **`waitpid`** waits for any child whose process group ID equals the absolute value of **`pid`**.

Third Parameter of `waitpid` Function

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

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- The **`options`** parameter is either **zero** or is constructed from the bitwise OR of the constants: **`WNOHANG`**, **`WUNTRACED`**, and **`WCONTINUED`**
- The **`options = 0`** :: The **`waitpid`** waits until the child change state.
- The option **`WNOHANG`** :: The **`waitpid`** will not block if a child specified by **`pid`** is not immediately available. In this case, the return value is 0.
- The option **`WUNTRACED`** :: The **`waitpid`** to report the status of unreported child processes that have been stopped. The **`WIFSTOPPED`** macro determines whether the return value corresponds to a stopped child process.
- The **`WCONTINUED`** :: The **`waitpid`** returns if a stopped child has been resumed by delivery of the signal **`SIGCONT`**.

Status Values: stat_loc Parameter of wait, & waitpid Functions

```
pid_t wait(int *stat_loc);

pid_t waitpid(pid_t pid, int *stat_loc, int options);
```

- ✍ The **stat_loc** argument of **wait** or **waitpid** is a pointer to an integer variable.
- ✍ If it is not **NULL**, these functions store the return status of the child in this location.
- ☞ The child returns its status by calling **exit**, **_exit**, **_Exit** or **return** from main.
- ☞ A zero return value indicates **EXIT_SUCCESS**; any other value indicates **EXIT_FAILURE**.
- ☞ The parent can only access the 8 least significant bits of the child's return status.

wait Vs waitpid a case

```
wait(NULL)          or          waitpid(-1, NULL, 0);
```

```
int status;

wait(&status)       or      waitpid(-1, &status, 0);
```

More Features of `waitpid` than `wait`

- ✍ The **`waitpid`** function lets us wait for one particular process, whereas the **`wait`** function returns the status of any terminated child.

- ✍ The **`waitpid`** function provides a nonblocking version of **`wait`**. There are times when we want to fetch a child's status, but we don't want to block.

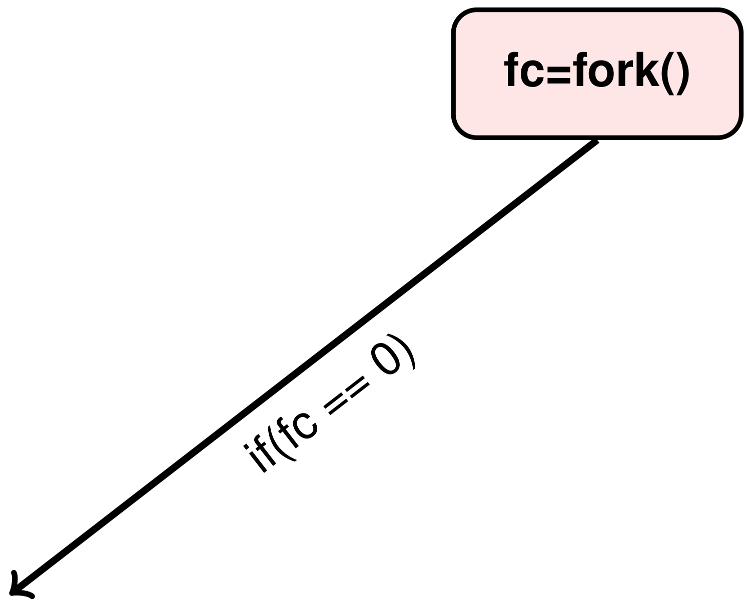
- ✍ The **`waitpid`** function provides support for job control with the **`WUNTRACED`** and **`WCONTINUED`** options.

Process Ordering Using `wait()` and `fork` return value

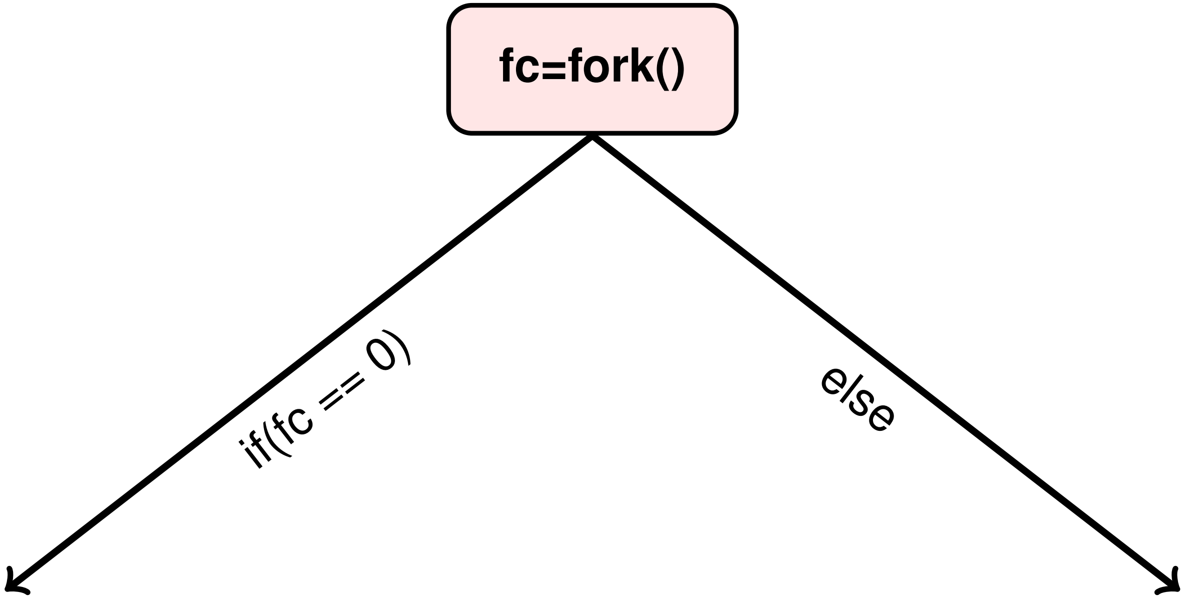
Process Ordering Using `wait()` and `fork` return value

fc=fork()

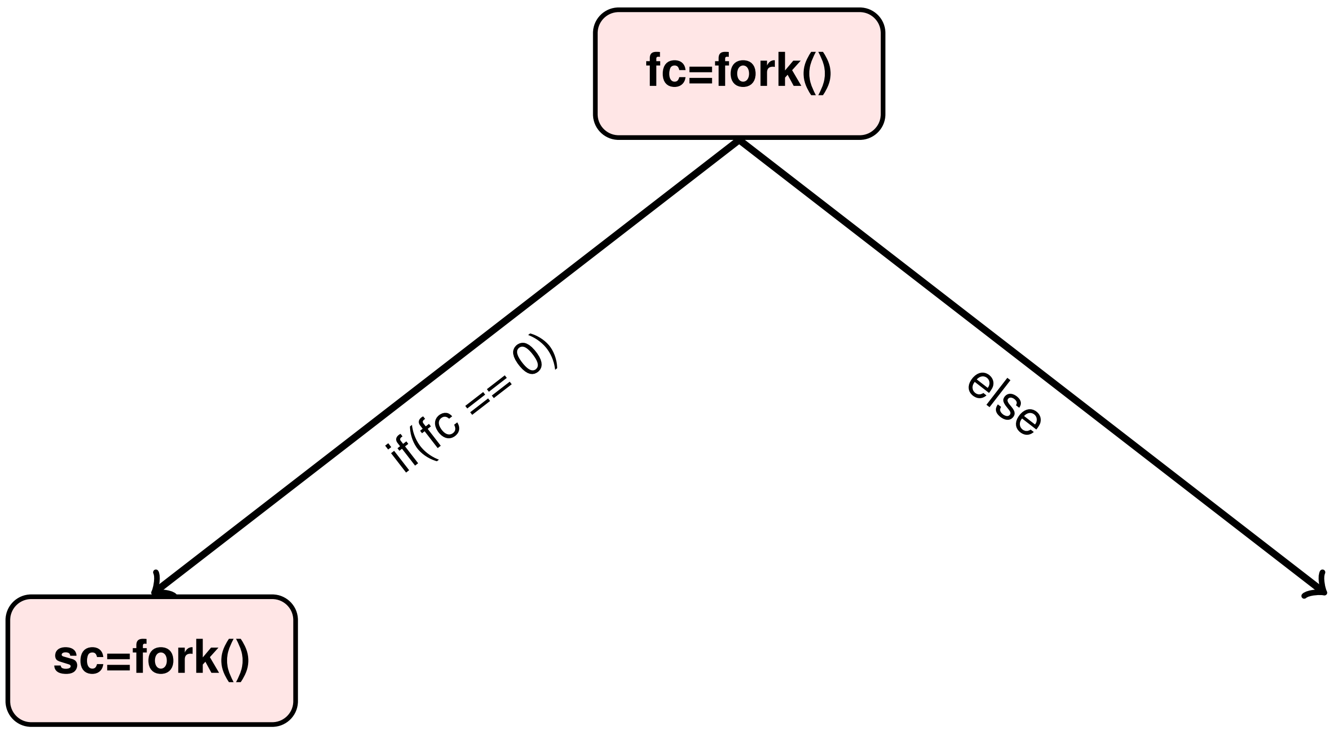
Process Ordering Using `wait()` and `fork` return value



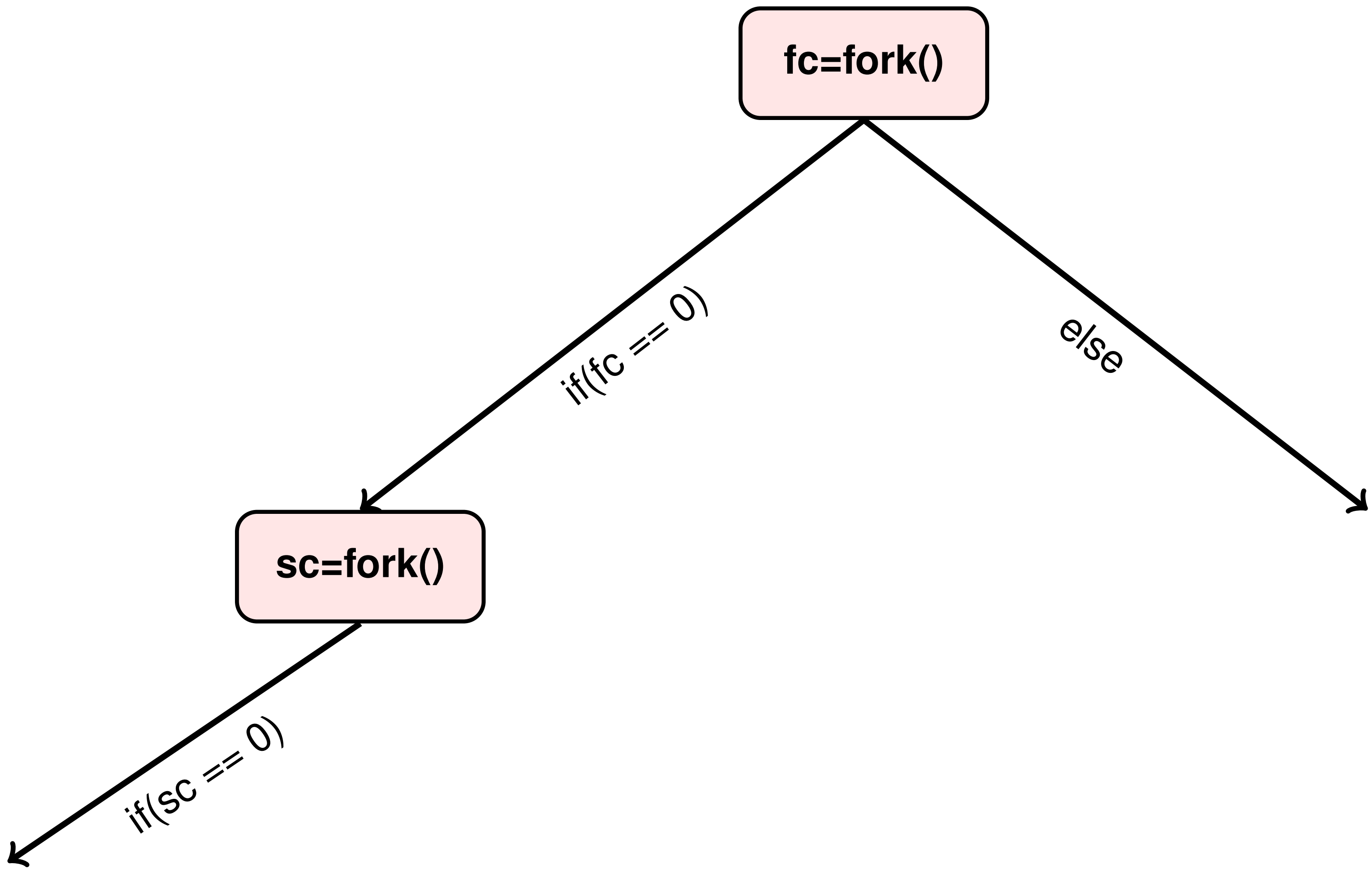
Process Ordering Using `wait()` and `fork` return value



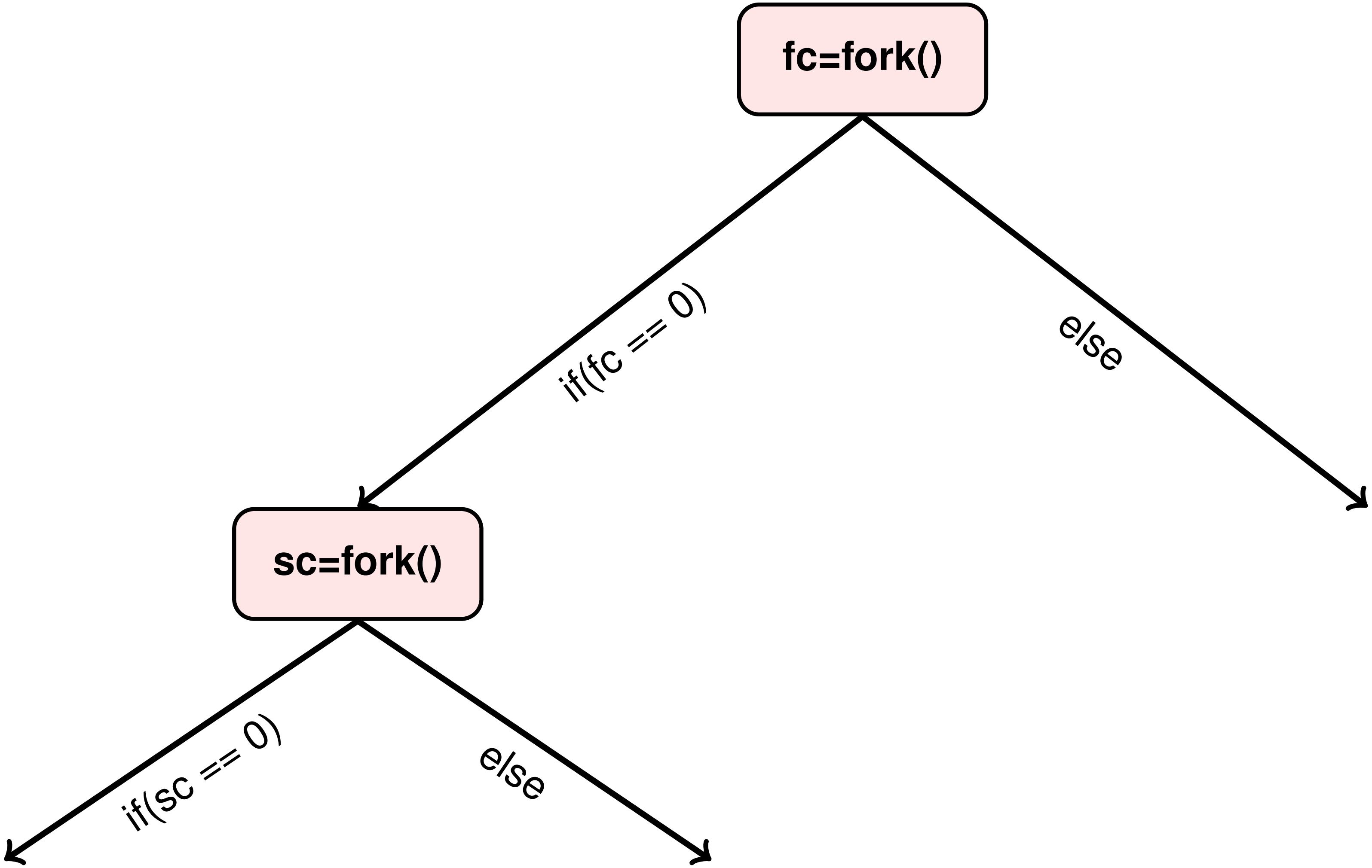
Process Ordering Using `wait()` and `fork` return value



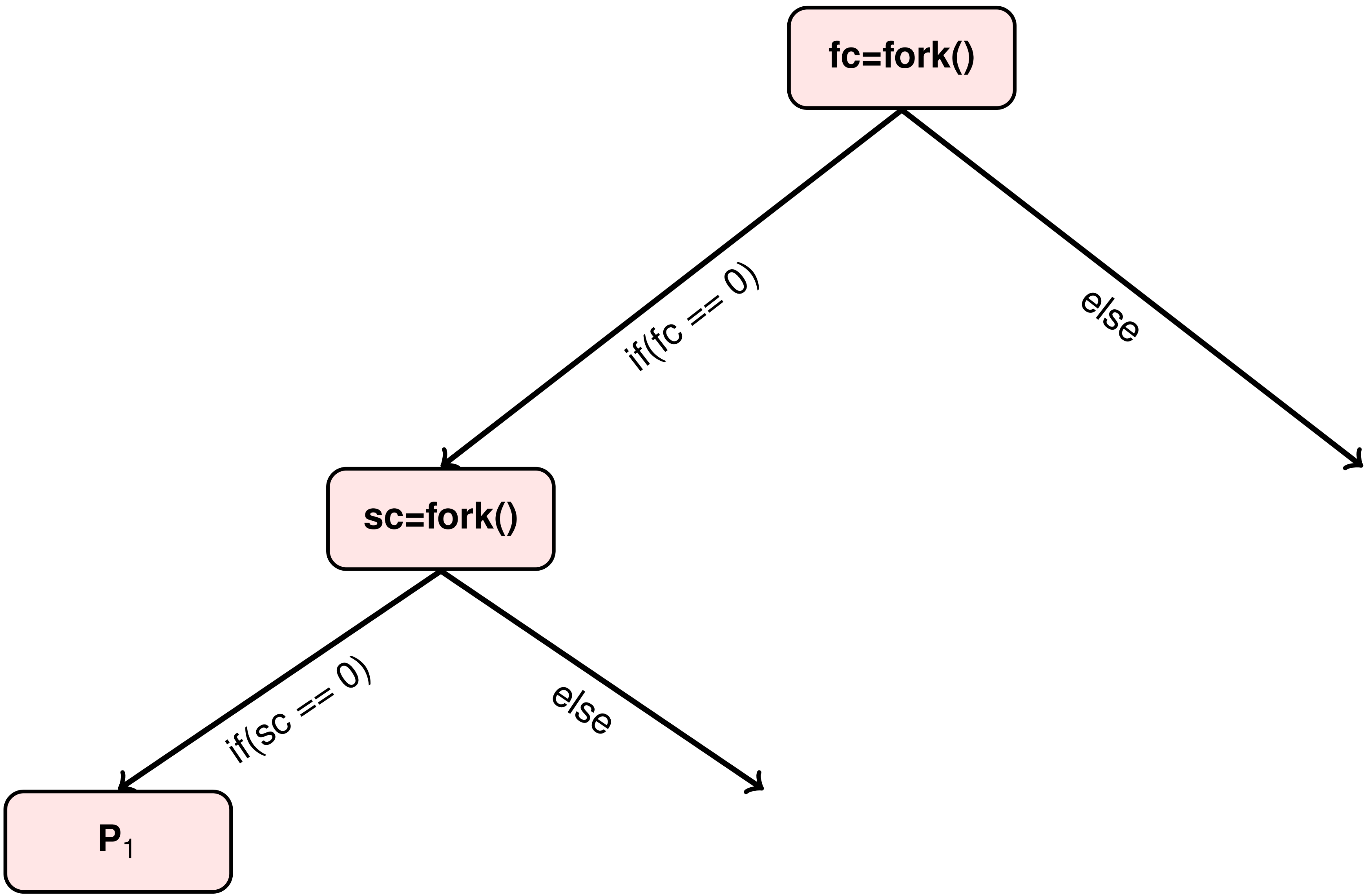
Process Ordering Using `wait()` and `fork` return value



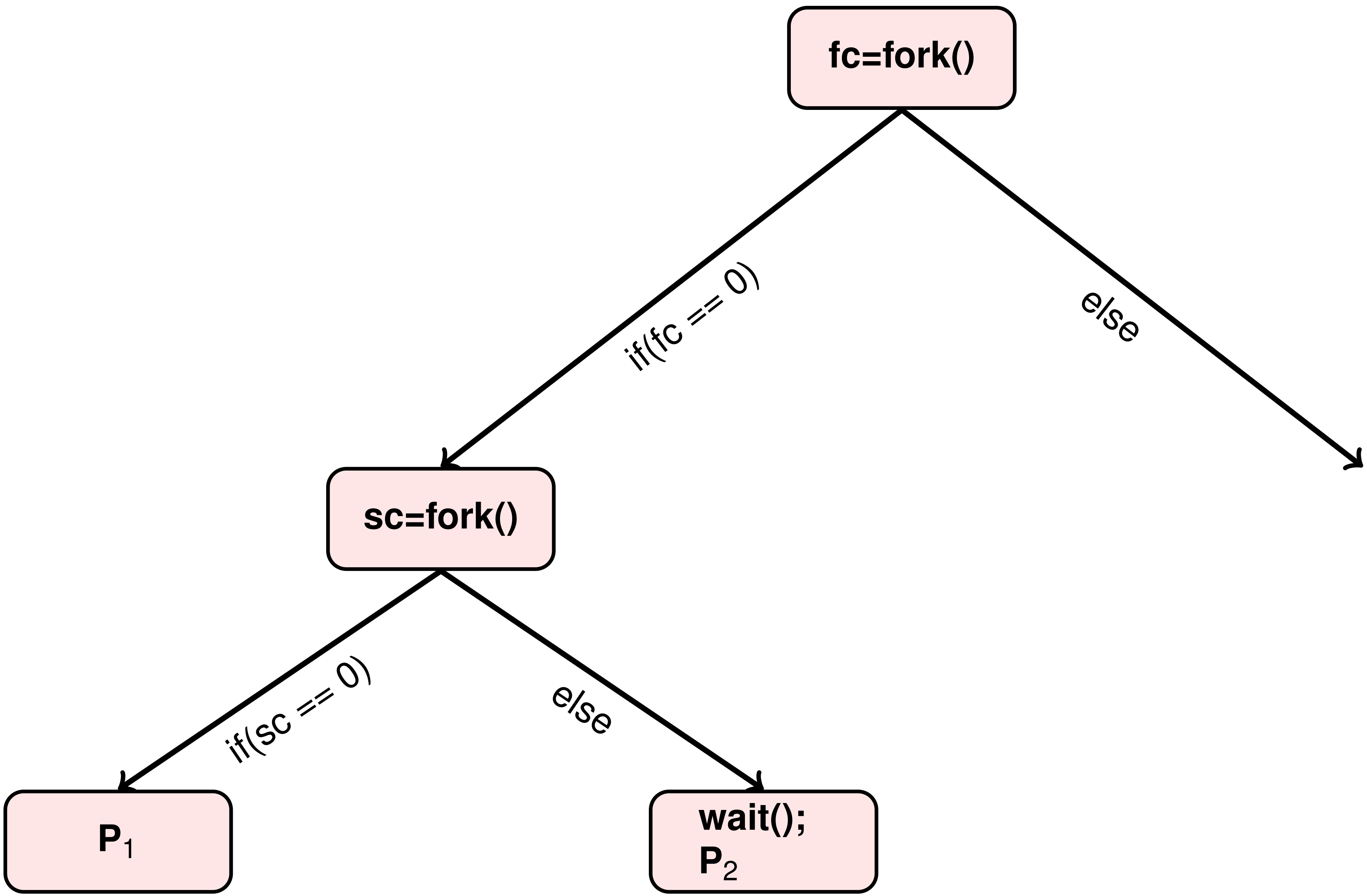
Process Ordering Using `wait()` and `fork` return value



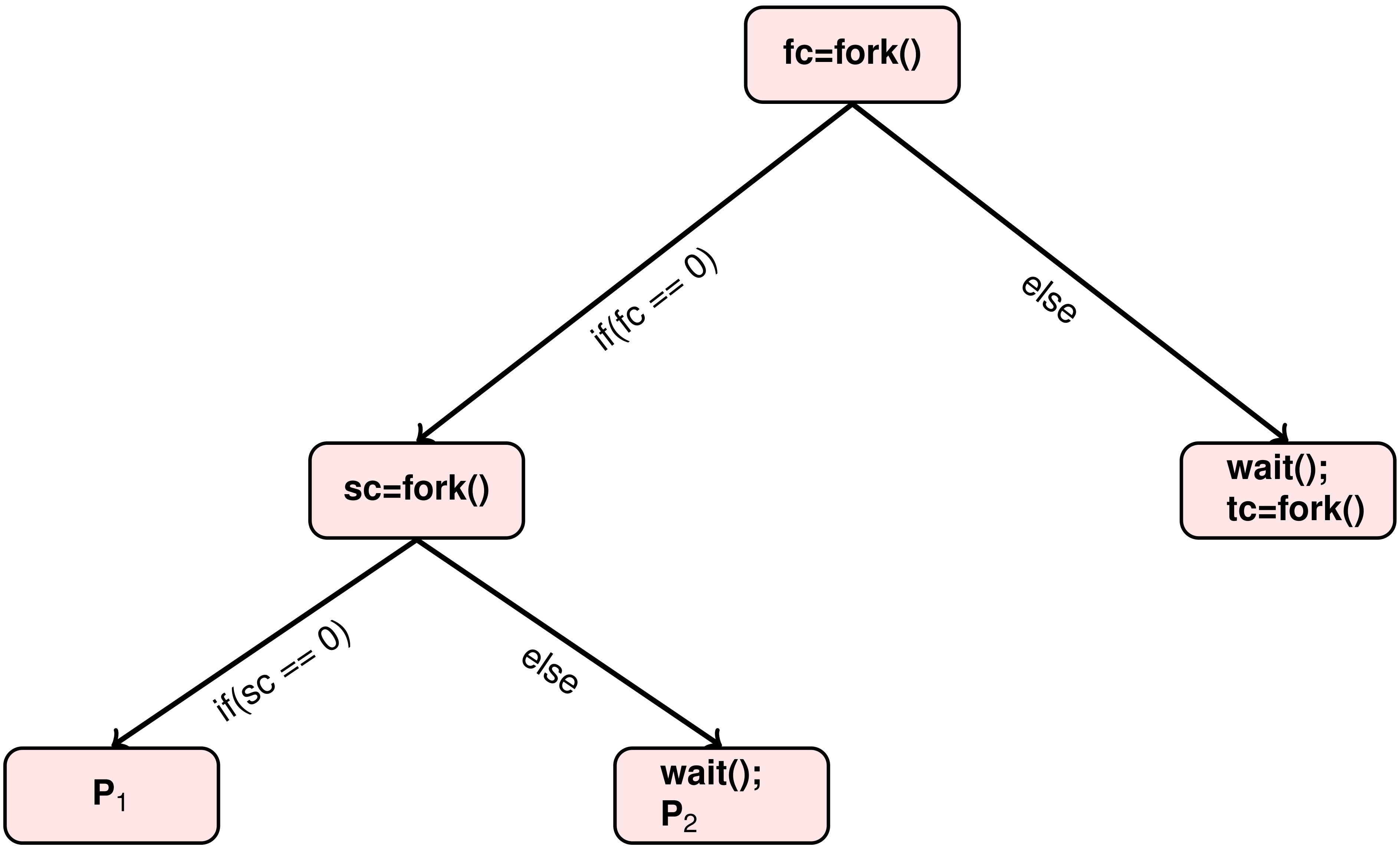
Process Ordering Using `wait()` and `fork` return value



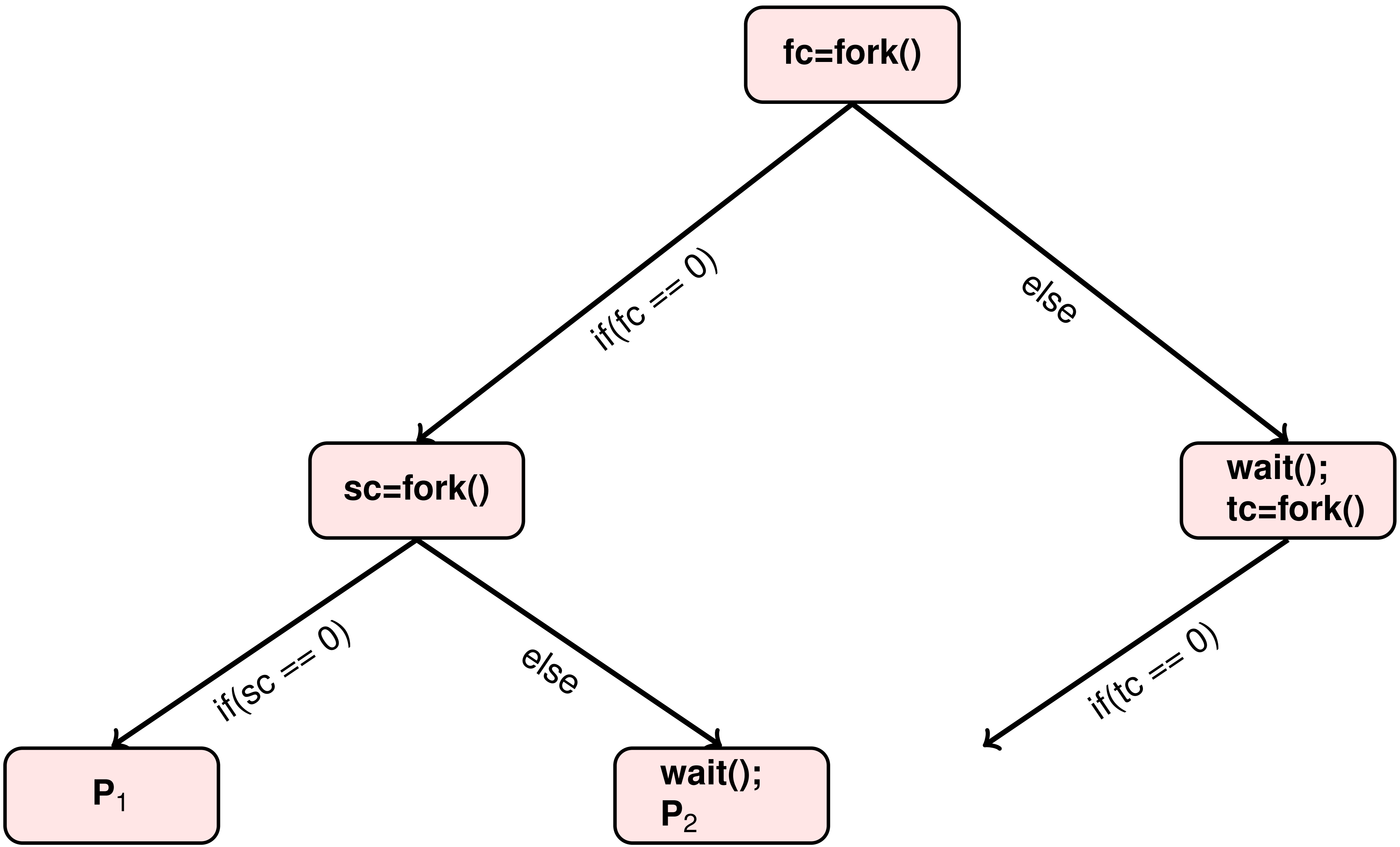
Process Ordering Using `wait()` and `fork` return value



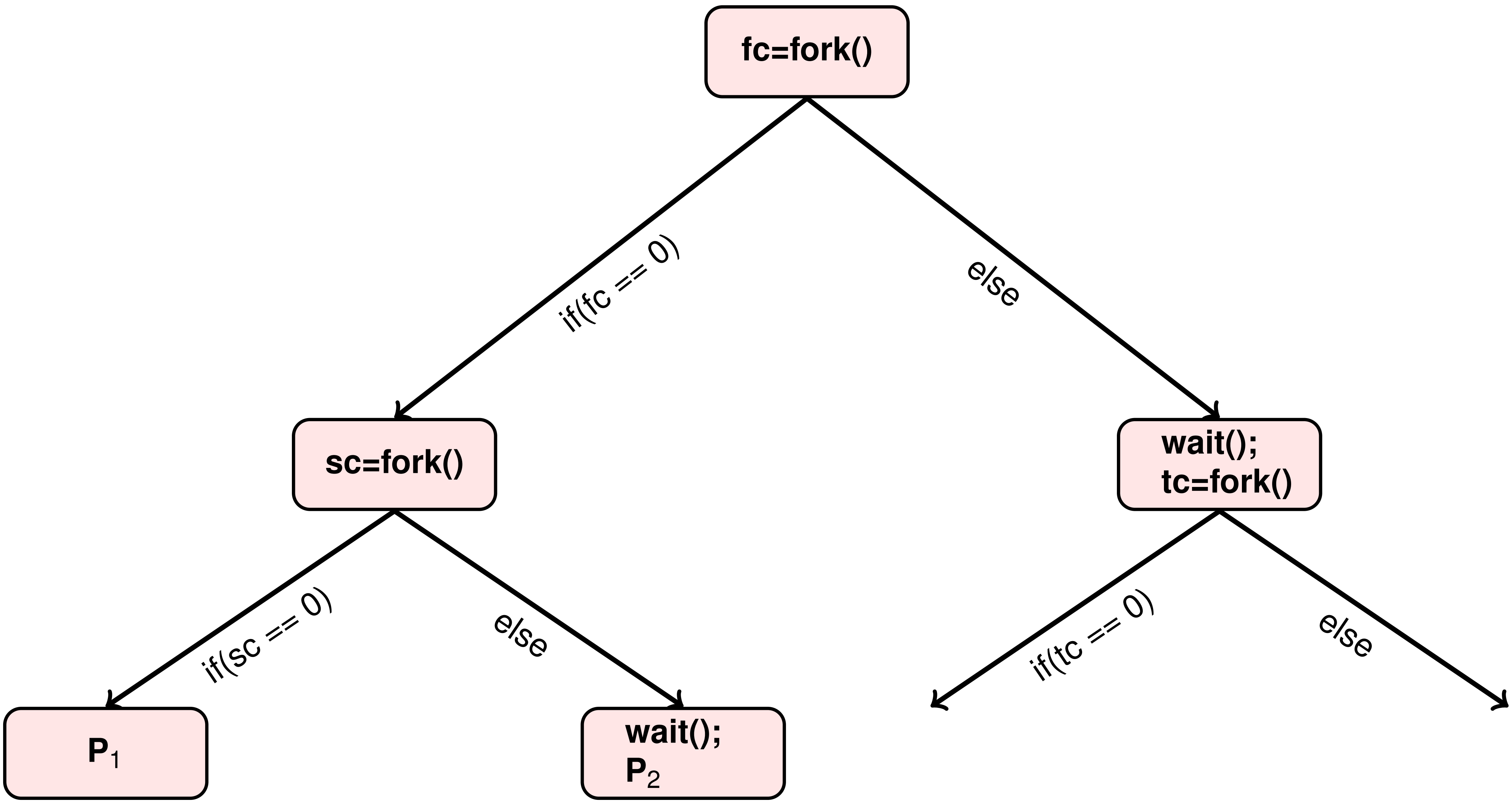
Process Ordering Using `wait()` and `fork` return value



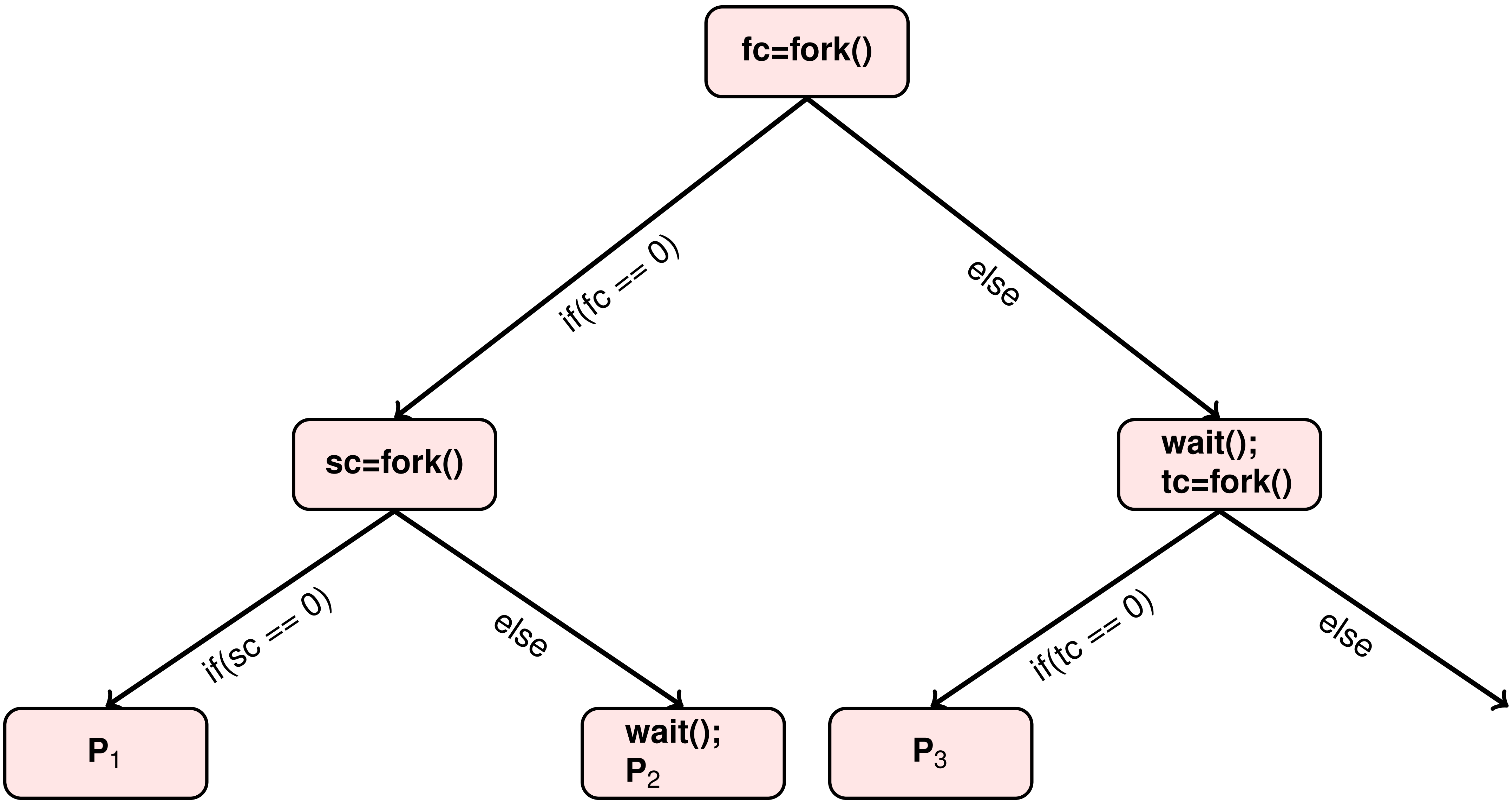
Process Ordering Using `wait()` and `fork` return value



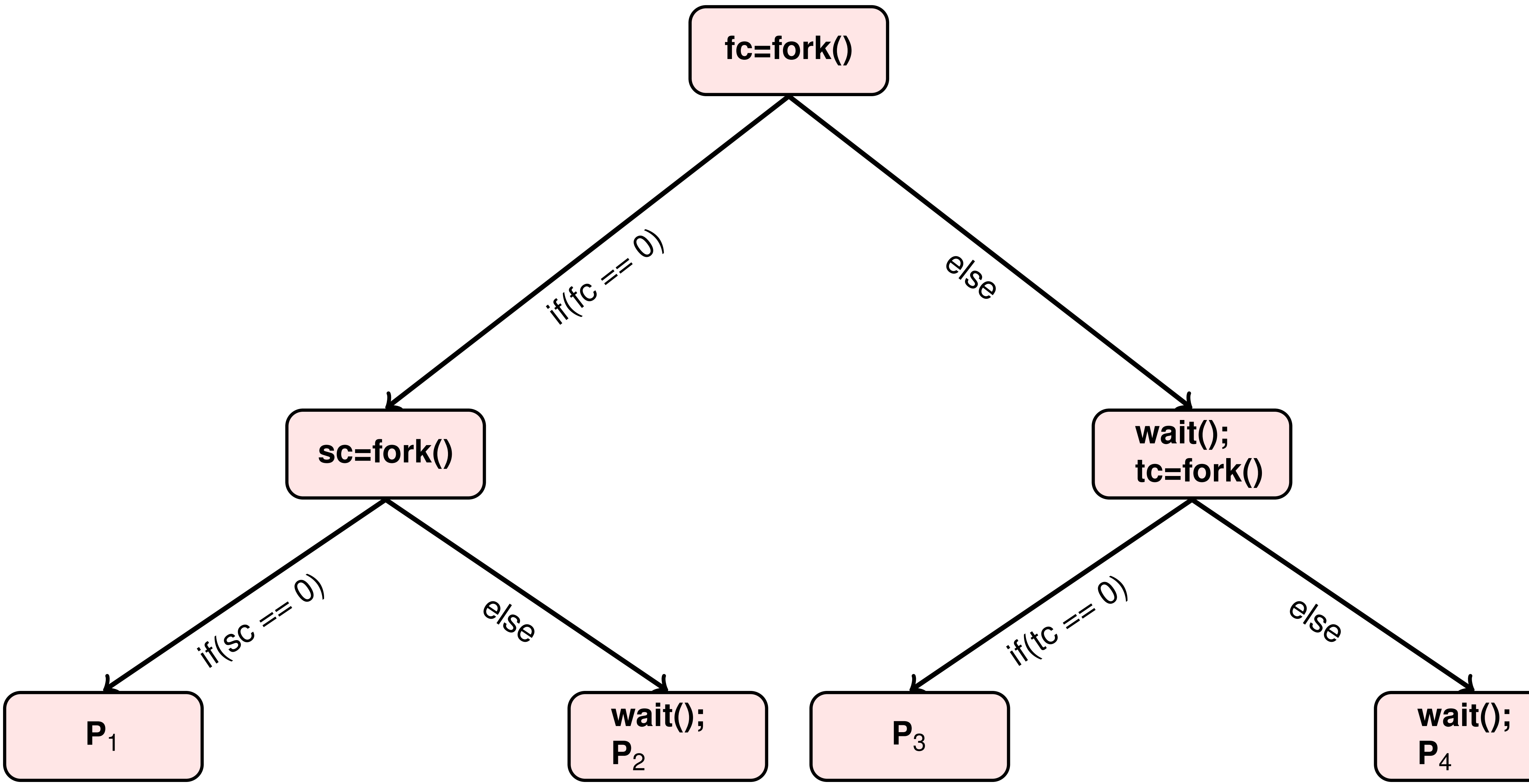
Process Ordering Using `wait()` and `fork` return value



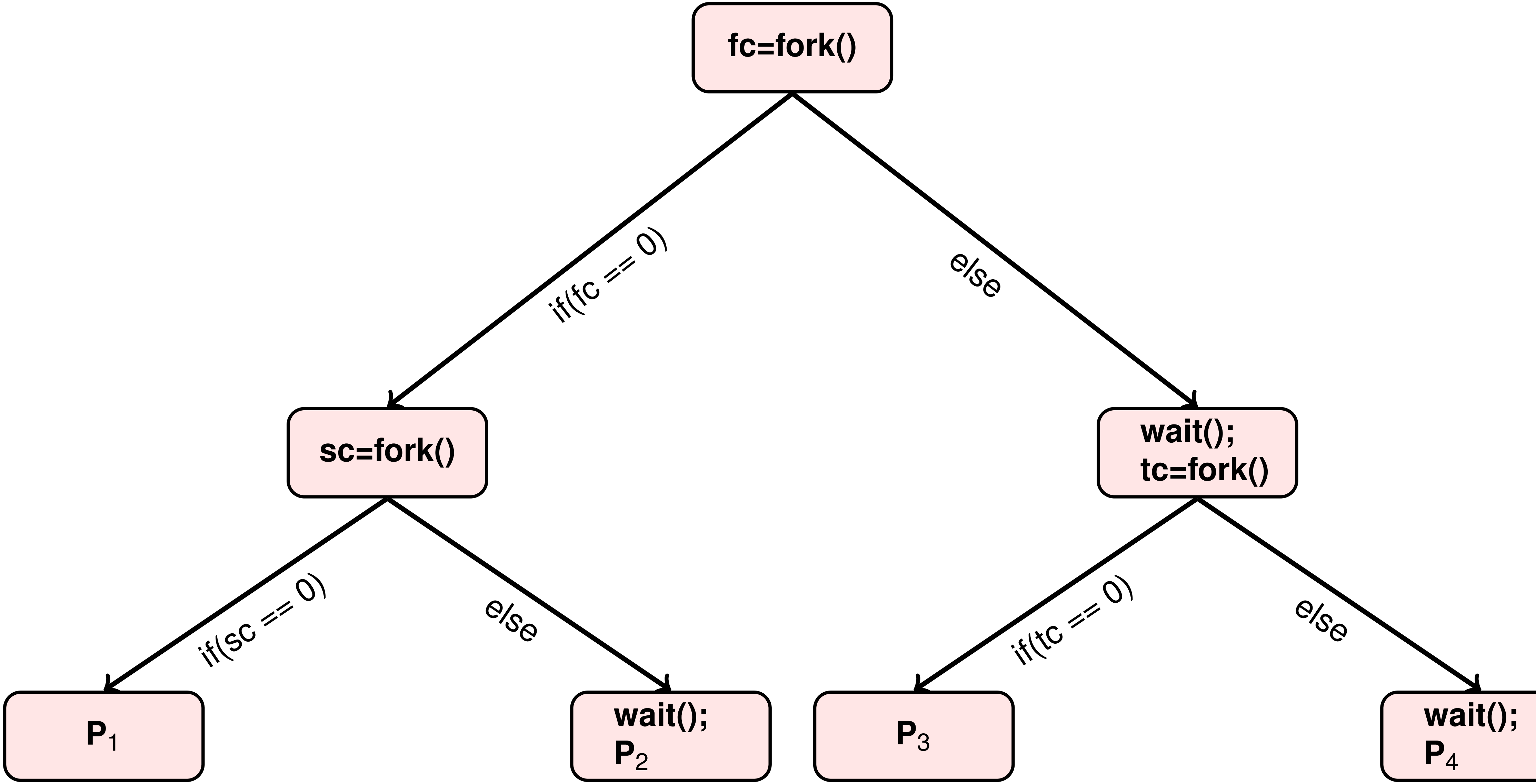
Process Ordering Using `wait()` and `fork` return value



Process Ordering Using `wait()` and `fork` return value



Process Ordering Using `wait()` and `fork` return value



Sample Code

```
int main(void){
  pid_t fc,sc,tc;
  fc=fork();
  if(fc==0){
      sc=fork();
      if(sc==0)
         printf("Process 1\n");
      else{
         wait(NULL);
         printf("Process 2\n");
      }
  }
  else{
       wait(NULL);
       tc=fork();
       if(tc==0)
          printf("Process 3\n");
       else{
           wait(NULL);
           printf("Process 4\n");
        }
   }
  return 0;
}
```

Sample Code

```
int main(void){
  pid_t fc,sc,tc;
  fc=fork();
  if(fc==0){
      sc=fork();
      if(sc==0)
         printf("Process 1\n");
      else{
         wait(NULL);
         printf("Process 2\n");
      }
  }
  else{
       wait(NULL);
       tc=fork();
       if(tc==0)
          printf("Process 3\n");
       else{
            wait(NULL);
            printf("Process 4\n");
        }
   }
  return 0;
}
```

☞ Extend the code:

✍ To test **`fork`** return value for -1.

✍ Under P_1 perform addition of two numbers, under P_2 for multiplication of two numbers, under P_3 perform sum of digits of a number, and under P_4 make the reverse of a number.

✍ Generalize for *n* processes if required.

The exec Family of System Calls

- ✍ The **`exec`** family of functions replaces the current process image with a new process image.
- ✍ The **`fork`** function creates a copy of the calling process, but many applications require the child process to execute code that is different from that of the parent.
- ✍ The **`exec`** family of functions provides a facility for overlaying the process image of the calling process with a new image.
- ✍ The **`exec`** operation replaces the entire address space(text, data, and stack) with that of the new process.
- ✍ Since the stack is also replaced, the call to **`exec`** family of functions donot return unless it results in an error.
- ✍ The child executes (with an **`exec`** function) the new program while the parent continues to execute the original code.
- ✍ All **`exec`** functions return -1 and set **`errno`** if unsuccessful.
- ✍ If any of these functions return at all, the call was unsuccessful.

The `exec` Family Series

- The **`exec: l`** series: **`l`** - a fixed list of arguments
  - **`execl(argument list)`**
  - **`execle(argument list)`**
  - **`execlp(argument list`**
- The **`exec: v`** series: **`v`** - a variable number of arguments
  - **`execv(argument list)`**
  - **`execve(argument list)`**
  - **`execvp(argument list)`**

The exec Family System Call Prototype

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg0, ... /*, char *(0) */);

int execle (const char *path, const char *arg0, ... /*, char *(0), char *const
      envp[] */);

int execlp (const char *file, const char *arg0, ... /*, char *(0) */);

int execv(const char *path, char *const argv[]);

int execve (const char *path, char *const argv[], char *const envp[]);

int execvp (const char *file, char *const argv[]);
```

Variations of the `exec` functions

- The six variations of the **`exec`** function differ in the way command-line arguments and the environment are passed.
- They also differ in whether a full pathname must be given for the executable.
- The **`execl`** (**`execl`**, **`execlp`** and **`execle`**) functions pass the command-line arguments in an explicit list and are useful if the number of command-line arguments are known at compile time.
- The **`execv`** (**`execv`**, **`execvp`** and **`execve`**) functions pass the command-line arguments in an argument array.
- The **`arg`**$_i$ parameter represents a pointer to a string.
- **`argv`** and **`envp`** represent **`NULL`**-terminated arrays of pointers to strings.
- The **`path`** parameter to **`execl`** is the pathname of a process image file specified either as a fully qualified pathname or relative to the current directory.
- The individual command-line arguments are then listed, followed by a **`(char *) 0`** pointer (a **`NULL`** pointer).

The `execl` System Call

```
#include <unistd.h>

int execl(const char *path, const char *arg0, ... /*, char *(0) */);

return -1 if unsuccessful
on successful no return to the calling process
```

Example: `execl`

A program that creates a child process to run **`ls -l`**.

```
int main(void) {
   pid_t childpid;
   childpid = fork();
   if (childpid == -1) {
      perror("Failed to fork");
      return 1;
   }
   if (childpid == 0) {                          /* child code */
      execl("/bin/ls", "ls", "-l", NULL);
      perror("Child failed to exec ls");
      return 1;
   }
   if (childpid != wait(NULL)) {                 /* parent code */
      perror("Parent failed to wait due to signal or error");
      return 1;
   }
   return 0;
}
```

The `execle` System Call

```
#include <unistd.h>

int execle (const char *path, const char *arg0, ... /*, char *(0),
    char *const envp[] */);

return -1 if unsuccessful
on successful no return to the calling process
```

Example: `execle`

A program that creates a child process to run **`wc`**.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void)
{
   int err;
   err=execle("/usr/bin/wc","wc","execldemo.c",NULL,NULL);
   if(err==-1){
   perror("Execle Failed\n");
}
  return 0;
}
```

The `execlp` System Call

```
#include <unistd.h>

int execlp (const char *file, const char *arg0, ... /*, char *(0) */
    );

return -1 if unsuccessful
on successful no return to the calling process
```

NOTE: The `execlp` System Call

- ✍ The first parameter is the name of the executable; **`file`**.
- ✍ **`execlp`** uses the `PATH` environment variable to search for the executable.
  (It is the similar way, how the shell tries to locate the executable file in one of the directories specified by the `PATH` variable when a user enters a command.)
- ✍ If the first parameter (**`file`**) contains a slash, then **`execlp`** treats file as a **pathname** and behaves like **`execl`**.

Example: `execlp`

A program that creates a child process to run **echo**.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void)
{
   char *temp1,*temp2;
   temp1="Funny"; temp2="world";
   pid_t pid;
   pid=fork();
   if(pid==0){
      execlp("echo","echo",temp1,temp2,NULL);
      printf("Error");
      return 1;
  }
  else{
      /* Parent code*/
   }
   return 0;
}
```

The **execv** System Call

- The **execv** function takes exactly two parameters, a pathname for the executable and an argument array.
- Used to run a command or executable with any number of arguments.
- Use an **execv** function with an argument array constructed at run time.
- The **execve** and **execvp** are variations on **execv**; they are similar in structure to execle and execlp, respectively.
- SYNOPSIS:

```
#include <unistd.h>

int execv(const char *path, char *const argv[]);

return -1 if unsuccessful
on successful no return to the calling process
```

Example: execv

A program that creates a child process to run **ls -l**.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void)
{
   char *cmdargs[]={"ls", "-l", NULL};
   pid_t pid;
   pid=fork();
   if(pid==0){
      execv("/bin/ls",cmdargs);
   }
   else{
      wait(NULL);
      printf("chile terminate\n");
   }
   return 0;
}
```

A Sample to Create 2nd Argument of `execv`

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc, char *argv[])
{
    int i;
    char *cmdarg[20];
    for(i=1;argv[i]!=NULL;i++){
        cmdarg[i-1]=argv[i];
    }
    cmdarg[i-1]=NULL;
    execv("/bin/ls",cmdarg);
    return 0;
}
```

- To run `./a.out ls`
- To run `./a.out ls -l`

The `execvp` System Call

```
#include <unistd.h>

int execvp (const char *file, char *const argv[]);

return -1 if unsuccessful
on successful no return to the calling process
```

Example: execvp

A program that creates a child process to run `ls -l`.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void)
{
   char *cmdargs[]={"ls", "-l", NULL};
   pid_t pid;
   pid=fork();
   if(pid==0){
      execvp("ls",cmdargs);
   }
   else{
      wait(NULL);
      printf("chile terminate\n");
   }
   return 0;
}
```

`execvp` to take arguments from Commandline

A program that creates a child process to run `ls -l`.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc, char *argv[])
{
   pid_t pid;
   pid=fork();
   if(pid==0){
      execvp("ls",&argv[1]);
   }
   else{
      wait(NULL);
      printf("chile terminate\n");
   }
   return 0;
}
```

`execvp` to take arguments from Commandline

A program that creates a child process to run **`ls -l`**.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(int argc, char *argv[])
{
   pid_t pid;
   pid=fork();
   if(pid==0){
      execvp(argv[1],&argv[1]);
   }
   else{
      wait(NULL);
      printf("chile terminate\n");
   }
   return 0;
}
```

The `execve` System Call

```
#include <unistd.h>

int execve (const char *path, char *const argv[],
    char *const envp[]);

return -1 if unsuccessful
on successful no return to the calling process
```

Example: execve

A program that creates a child process to run **ls -l**.

```
#include<stdio.h>
#include<stdlib.h>
#include<unistd.h>

int main(void)
{
   char *cmdargs[]={"ls", "-l", NULL};
   pid_t pid;
   pid=fork();
   if(pid==0){
      execve("/bin/ls", cmdargs, NULL);
   }
   else{
      wait(NULL);
      printf("chile terminate\n");
   }
   return 0;
}
```

Process Environment

- An *environment list* consists of an array of pointers to strings of the form **`name = value`**.
- The **`name`** specifies an environment variable, and the **`value`** specifies a string value associated with the environment variable.
- The last entry of the array is **`NULL`**.
- The **external** variable **`environ`** points to the process environment list when the process begins executing. The strings in the process environment list can appear in any order.
- SYNOPSIS:

```
#include<stio.h>

extern char **environ;
```

NOTE::

- If the process is initiated by **`execl`**, **`execlp`**, **`execv`** or **`execvp`**, then the process inherits the environment list of the process just before the execution of **`exec`**.
- The **`execle`** and **`execve`** functions specifically set the environment list.

Program to Print Environment List

```c
#include <stdio.h>

extern char **environ;

int main(void) {
   int i;
   printf("The environment list follows:\n");
   for(i = 0; environ[i] != NULL; i++){
     sleep(2);
     printf("environ[%d]: %s\n", i, environ[i]);
   }
   return 0;
}
```

Background Processes and Daemons

- The shell is a command interpreter that prompts for commands, reads the commands from standard input, forks children to execute the commands and waits for the children to finish.
- When standard input and output come from a terminal type of device, a user can terminate an executing command by entering the interrupt character(CTRL+C).
- Most shells interpret a line ending with & as a command that should be executed by a background process.
- When a shell creates a background process, it does not wait for the process to complete before issuing a prompt and accepting additional commands.

Daemon

- A daemon is a background process that normally runs indefinitely.
- The UNIX operating system relies on many daemon processes to perform routine tasks.