Operating System Concepts

Lecture 6: Process Control

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

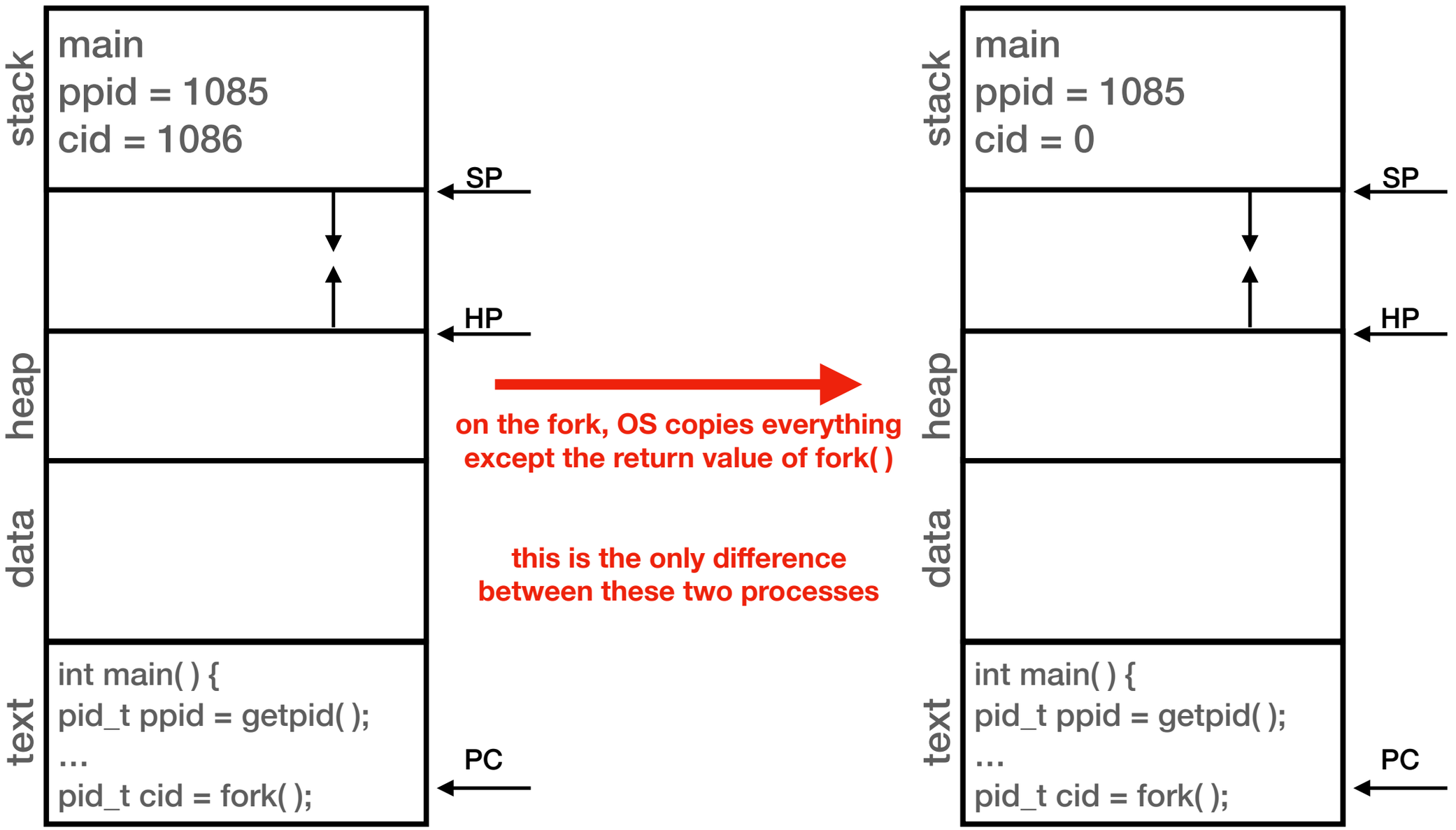
MWF 12:00-12:50 VVC 2 215

Today's class

- Process Control
  - How to create a new process?
  - How to terminate a process?
- Examples

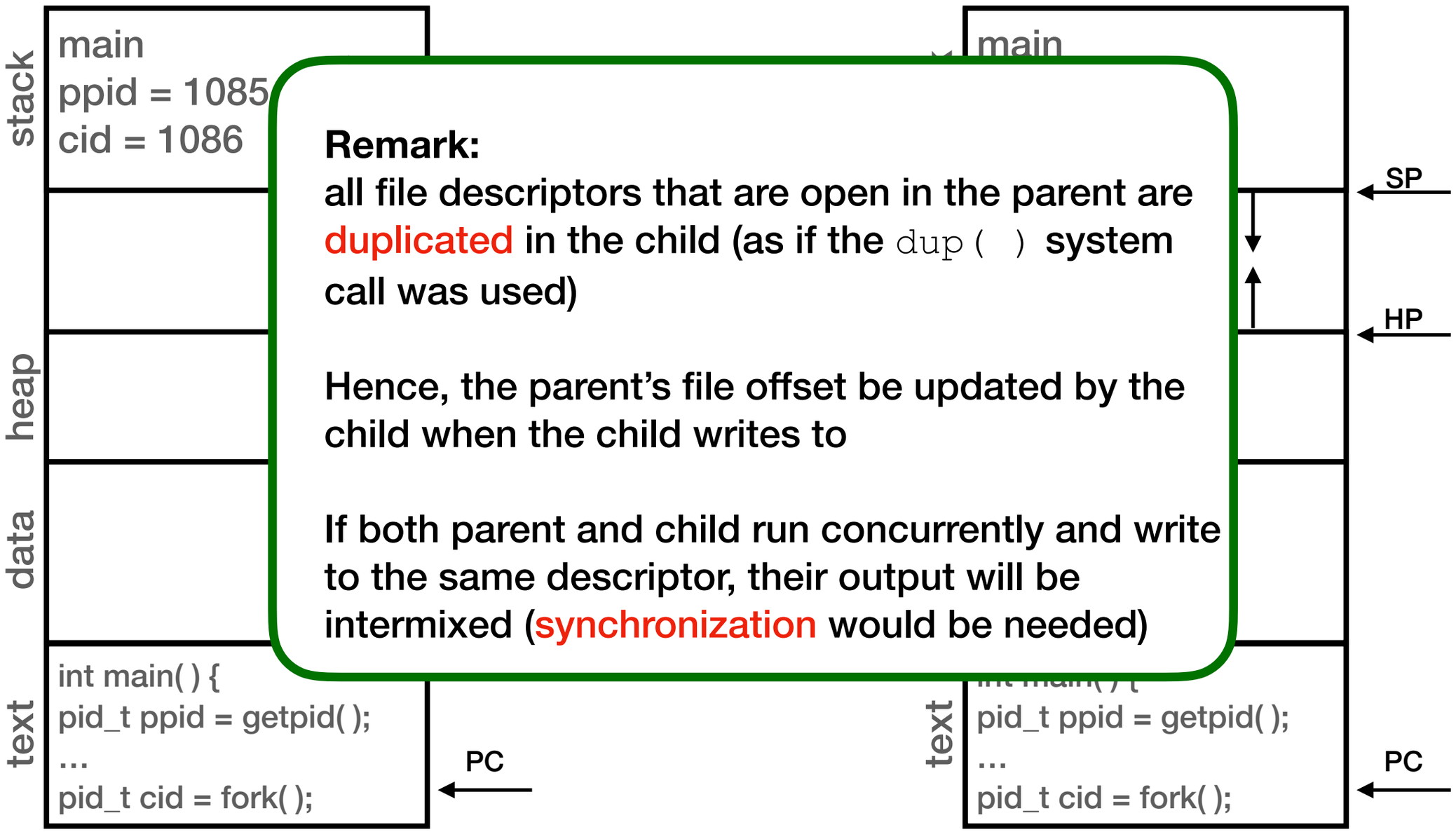
What happens on a fork?

```
pid_t cid = fork( );
```



What happens on a fork?

```
pid_t cid = fork( );
```



Terminating a process

- Process termination is the ultimate resource reclamation by the OS
  - closes all open files, connections, etc.
  - deallocates memory and most of the OS structures supporting the process
  - checks if parent is alive
    - if so, holds the **exit status** until parent requests it; in this case, process does not really die, but it enters the zombie state (**WHY?**)
    - If not, it deallocates all data structures; the process is dead at this point
  - cleans up all waiting zombies

Normal and abnormal termination

- a process can terminate normally by returning from `main`, or directly calling the standard C library function `exit( )` or the system call `_exit( );`
  When the `main` function returns, `exit( )` is called indirectly
  - `exit( )` calls all exit handlers that have been registered using `atexit( )` — a glibc function
  - `_exit( )` doesn't call exit handlers

Normal and abnormal termination

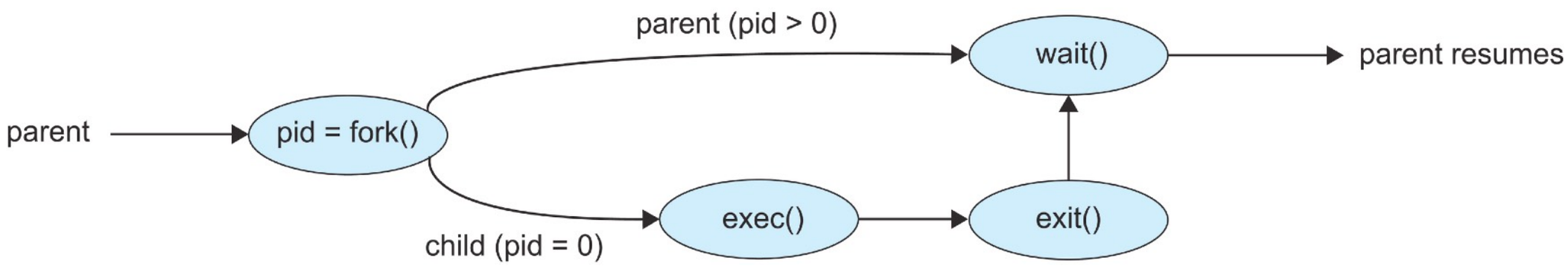
- a process can terminate normally by returning from `main`, or directly calling the standard C library function `exit( )` or the system call `_exit( );`
  When the `main` function returns, `exit( )` is called indirectly
  - `exit( )` calls all exit handlers that have been registered using `atexit( )` — a glibc function
  - `_exit( )` doesn't call exit handlers
- for abnormal termination of a process we can call `abort( )` which generates `SIGABRT`
  - functions registered using `atexit( )` are not called
  - it may not close open files or flush stream buffers

Normal and abnormal termination

- a process can terminate normally by returning from `main`, or directly calling the standard C library function `exit( )` or the system call `_exit( );`
  When the `main` function returns, `exit( )` is called indirectly
  - `exit( )` calls all exit handlers that have been registered using `atexit( )` — a glibc function
  - `_exit( )` doesn't call exit handlers
- for abnormal termination of a process we can call `abort( )` which generates `SIGABRT`
  - functions registered using `atexit( )` are not called
  - it may not close open files or flush stream buffers
- a process can terminate a child using the `kill( )` system call
  - `kill(cid, SIGKILL)`

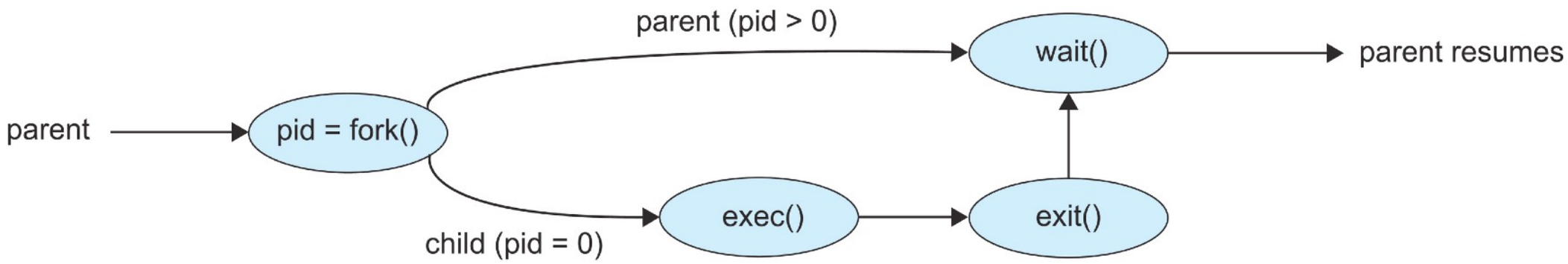
Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate



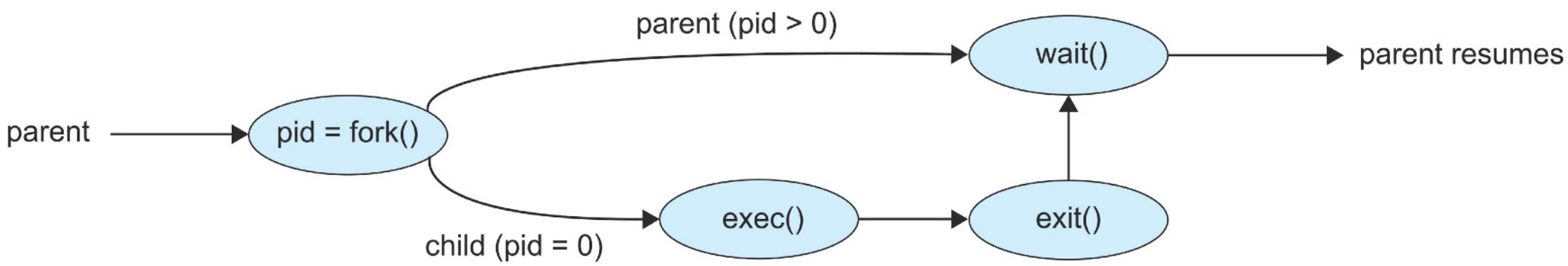
Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value



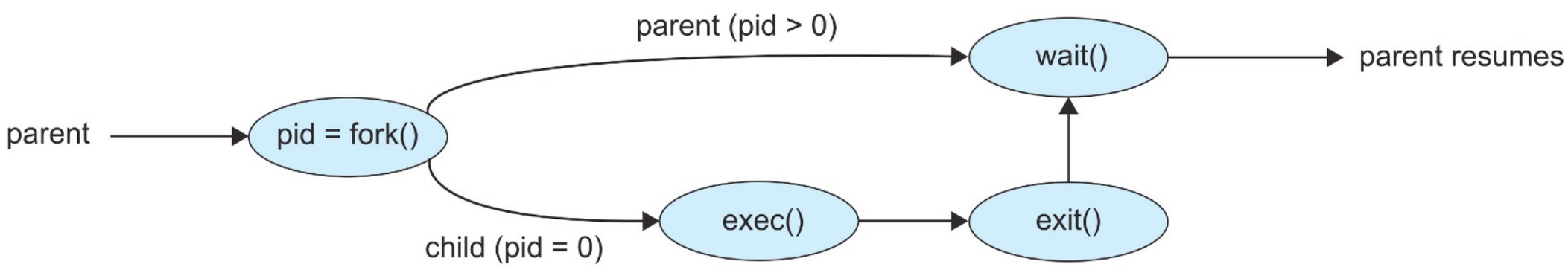
Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result



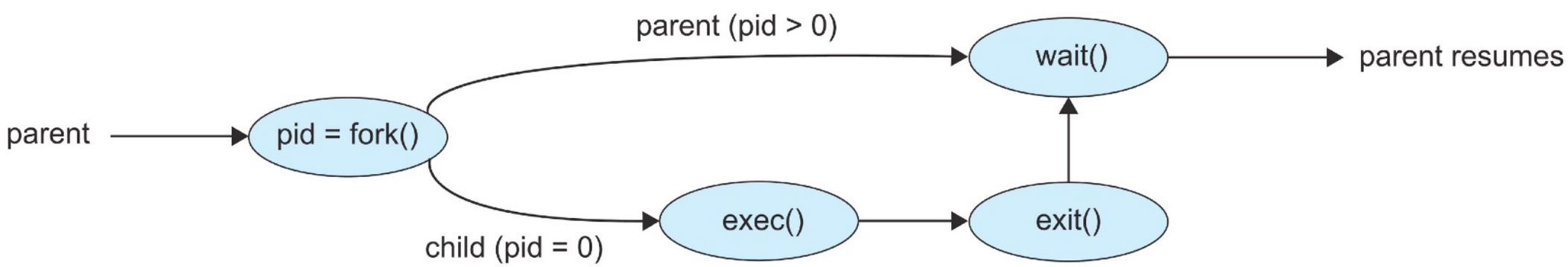
Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent; this unblocks the parent and returns the child's exit value along with the child's PID



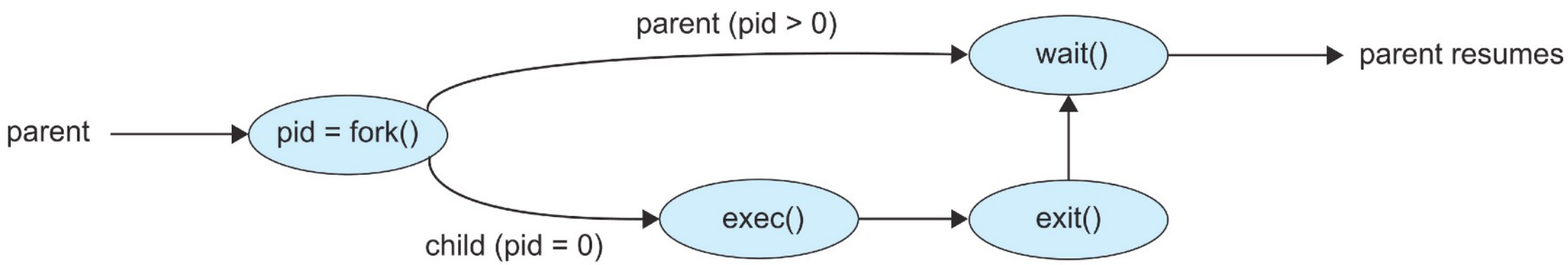
Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent; this unblocks the parent and returns the child's exit value along with the child's PID
  - if there are no children alive, `wait( )` returns immediately



Waiting for the child process to terminate

- the parent can execute concurrently with its children or can wait until some or all of them terminate
- the `wait( )` system call enables the parent process to wait for the child process to terminate and receive its return value
  - it puts the parent to sleep waiting for a child's result
  - when a child calls `exit( )`, the kernel notifies the parent by sending the `SIGCHLD` signal to the parent; this unblocks the parent and returns the child's exit value along with the child's PID
  - if there are no children alive, `wait( )` returns immediately
  - also, if there are zombies waiting for their parents, `wait( )` returns one of the exit statuses immediately (and deallocates the zombie)



Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z
- a process becomes orphan when its parent terminates while it is still running

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z
- a process becomes orphan when its parent terminates while it is still running
  - hence the parent terminates without invoking the `wait( )` system call

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z
- a process becomes orphan when its parent terminates while it is still running
  - hence the parent terminates without invoking the `wait( )` system call
  - UNIX and Linux systems assign the `init` process (PID=1) as the new parent of the orphan process (aka **reparenting**)

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z
- a process becomes orphan when its parent terminates while it is still running
  - hence the parent terminates without invoking the `wait( )` system call
  - UNIX and Linux systems assign the `init` process (PID=1) as the new parent of the orphan process (aka **reparenting**)
    - you can check this `if(getppid( ) == 1)`

Zombie and orphan processes

- a process that has terminated, but its parent has not (yet) called `wait( )` becomes zombie
  - the `ps` command prints the state of a zombie process as Z
- a process becomes orphan when its parent terminates while it is still running
  - hence the parent terminates without invoking the `wait( )` system call
  - UNIX and Linux systems assign the `init` process (PID=1) as the new parent of the orphan process (aka **reparenting**)
    - you can check this `if(getppid( ) == 1)`
  - the `init` process periodically calls `wait( )` allowing the exit status of any orphaned process to be collected and their process table entries be deleted

Fork example

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

int main() {
      ...
      pid_t ppid = getpid();                  // store parent's pid
      pid_t pid = fork();                     // create a child
      if(pid == 0){                           // child continues here
        printf("Child pid: [%d]\n", getpid());
        ...
      } else if (pid > 0) {                   // parent continues here
        printf("Parent pid: [%d] Child pid: [%d]\n", ppid, pid);
        ...
      } else {
        perror("fork failed!");
        exit(1);
      }
      …
}
```

Fork example

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

int main() {
      ...
      pid_t ppid = getpid();                    // store parent's pid
      pid_t pid = fork();                       // create a child
      if(pid == 0){                             // child continues here
        printf("Child pid: [%d]\n", getpid());
        ...
      } else if (pid > 0) {                     // parent continues here
        printf("Parent pid: [%d] Child pid: [%d]\n", ppid, pid);
        ...
      } else {
        perror("fork failed!");
        exit(1);
      }
      …
}
```

Run the `ps` command to check the processes' IDs

Combining fork and wait

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

int main() {
      ...
      pid_t ppid = getpid();                    // store parent's pid
      pid_t pid = fork();                       // create a child
      if(pid == 0){                             // child continues here
        ...
      } else if (pid > 0) {                     // parent continues here
        …
        pid_t cpid = wait(&child_status);
      } else {
        perror("fork failed!");
      }
      …
}
```

Combining fork and exec

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

main() {
      ...
      pid_t ppid = getpid();              // store parent's pid
      pid_t pid = fork();                 // create a child
      if(pid == 0){                       // child continues here
         status = execl("/bin/ls", arg0, arg1, …); // mark the end with a null pointer
        /* exec doesn't return when it works.
            so if we got here, it must have failed! */
        perror("exec failed!");
      } else if (pid > 0) {               // parent continues here
        ...
        cpid = wait(&status);             // pass NULL if not interested in exit status
        if (WIFEXITED(status))
            printf("child exit status was %d\n", WEXITSTATUS(status));
      } else {
        perror("fork failed!");
      }
      ...
}
```

Parent can kill its child!

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

main() {
      ...
      int ppid = getpid();                  // store parent's pid
      int pid = fork();                     // create a child
      if(pid == 0){                         // child continues here
            sleep(10);                      // child sleeps for 10 seconds
            ...
            exit(0);
      }
      else {                                // parent continues here
        ...
        printf( "Type any character to kill the child.\n" );
        char answer[10];
        gets(answer);
        if ( !kill(pid, SIGKILL) ) {
          printf("Killed the child.\n");
        }
      }
}
```

Other system calls for process control

- OS must include calls to enable special control of a process:

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value
  - Debugging support:

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value
  - Debugging support:
    - the `ptrace( )` system call allows a process to be put under control of another process

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value
  - Debugging support:
    - the `ptrace( )` system call allows a process to be put under control of another process
    - the other process can check the arguments of the system call made by the process being traced, set breakpoints, examine registers, etc.

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value
  - Debugging support:
    - the `ptrace( )` system call allows a process to be put under control of another process
    - the other process can check the arguments of the system call made by the process being traced, set breakpoints, examine registers, etc.
  - Alarms and time:

Other system calls for process control

- OS must include calls to enable special control of a process:
  - Priority manipulation:
    - the `nice(incr)` system call adjusts the process priority by adding `incr` to its nice value
      - lower nice values have higher scheduling priority
      - a process could be “nice” and reduce its share of the CPU by adjusting its nice value
  - Debugging support:
    - the `ptrace( )` system call allows a process to be put under control of another process
    - the other process can check the arguments of the system call made by the process being traced, set breakpoints, examine registers, etc.
  - Alarms and time:
    - the `sleep( )` system call puts a process on a timer queue waiting for some number of seconds, supporting alarm functionality

Process termination in UNIX systems

- the `kill` command sends an arbitrary signal to processes based on PID

Process termination in UNIX systems

- the `kill` command sends an arbitrary signal to processes based on PID
  - it sends a `SIGTERM` signal by default

Process termination in UNIX systems

- the `kill` command sends an arbitrary signal to processes based on PID
  - it sends a `SIGTERM` signal by default
- the `killall` command sends an arbitrary signal to processes based on process name

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes
  - `ps -el` lists complete information about all processes that are currently active in the system

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes
  - `ps -el` lists complete information about all processes that are currently active in the system
  - `ps -u [username]` lists all processes created by a specific user

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes
  - `ps -el` lists complete information about all processes that are currently active in the system
  - `ps -u [username]` lists all processes created by a specific user
- `top` provides a dynamic real-time view of a running system (repetitive update on active processes)

Process monitoring in UNIX systems

- `ps` displays information about a selection of the active processes
  - `ps -el` lists complete information about all processes that are currently active in the system
  - `ps -u [username]` lists all processes created by a specific user
- `top` provides a dynamic real-time view of a running system (repetitive update on active processes)
- `pstree` displays a tree of processes

Shell

- act as a process control system
  - allows programmer to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have shells

Shell

- act as a process control system
  - allows programmer to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have shells
- when you log in to a machine running UNIX, you create a shell process

Shell

- act as a process control system
  - allows programmer to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have shells
- when you log in to a machine running UNIX, you create a shell process
- every command launched in the shell is a child of the shell process and is an implicit `fork( )` and `exec( )` pair

Shell

- act as a process control system
  - allows programmer to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have shells
- when you log in to a machine running UNIX, you create a shell process
- every command launched in the shell is a child of the shell process and is an implicit `fork( )` and `exec( )` pair
- the separation of `fork( )` and `exec( )` enables features like input/output redirection, pipes, etc.
  - the shell runs code after the call to `fork( )` and before the call to `exec( )`

Summary

- OS creates, deletes, suspends, and resumes processes
- OS allocates resources to active processes
  - memory, I/O devices, files
- OS schedules processes
  - context switches between them
- OS supports **interprocess communication** and provides **synchronization** mechanisms

Homework

- Compile and run the examples available on eClass