Operating System Concepts

Lecture 5: Process Control

Omid Ardakanian
oardakan@ualberta.ca
University of Alberta

Today's class

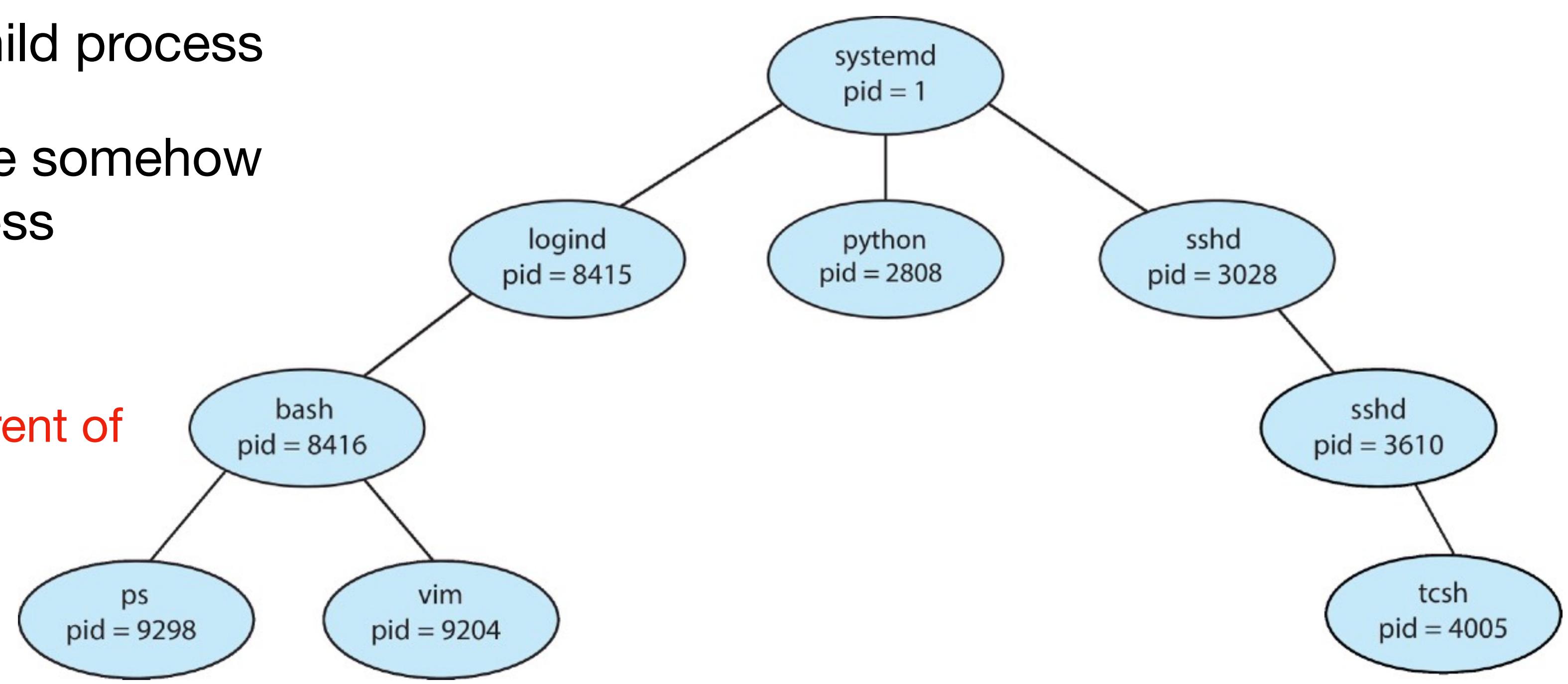
- Process Control
  - How to create a process?
  - How to terminate a process?
- Examples

Process management system calls in UNIX

- `getpid( )` returns the current PID
- `fork( )` copies the current process (a new PID is assigned to the child process)
- `execve( )` loads a new binary file into memory (without changing its PID)
- `wait( )` waits until one of its child processes terminates
- `waitpid( )` waits until the specified child process terminates
- `_exit( )` terminates a process
- `pause( )` causes the calling process to sleep until a signal is delivered that either terminates the process or causes the invocation of a signal-catching function
- `nanosleep( )` suspends execution of a process for at least the specified time or the delivery of a signal
- `kill( )` sends a signal (interrupt-like notification) to another process (if permitted)
- `sigaction( )` sets handlers for signals except `SIGKILL` and `SIGSTOP`
  - these signals cannot be caught, blocked, or ignored

Creating a process with the fork system call

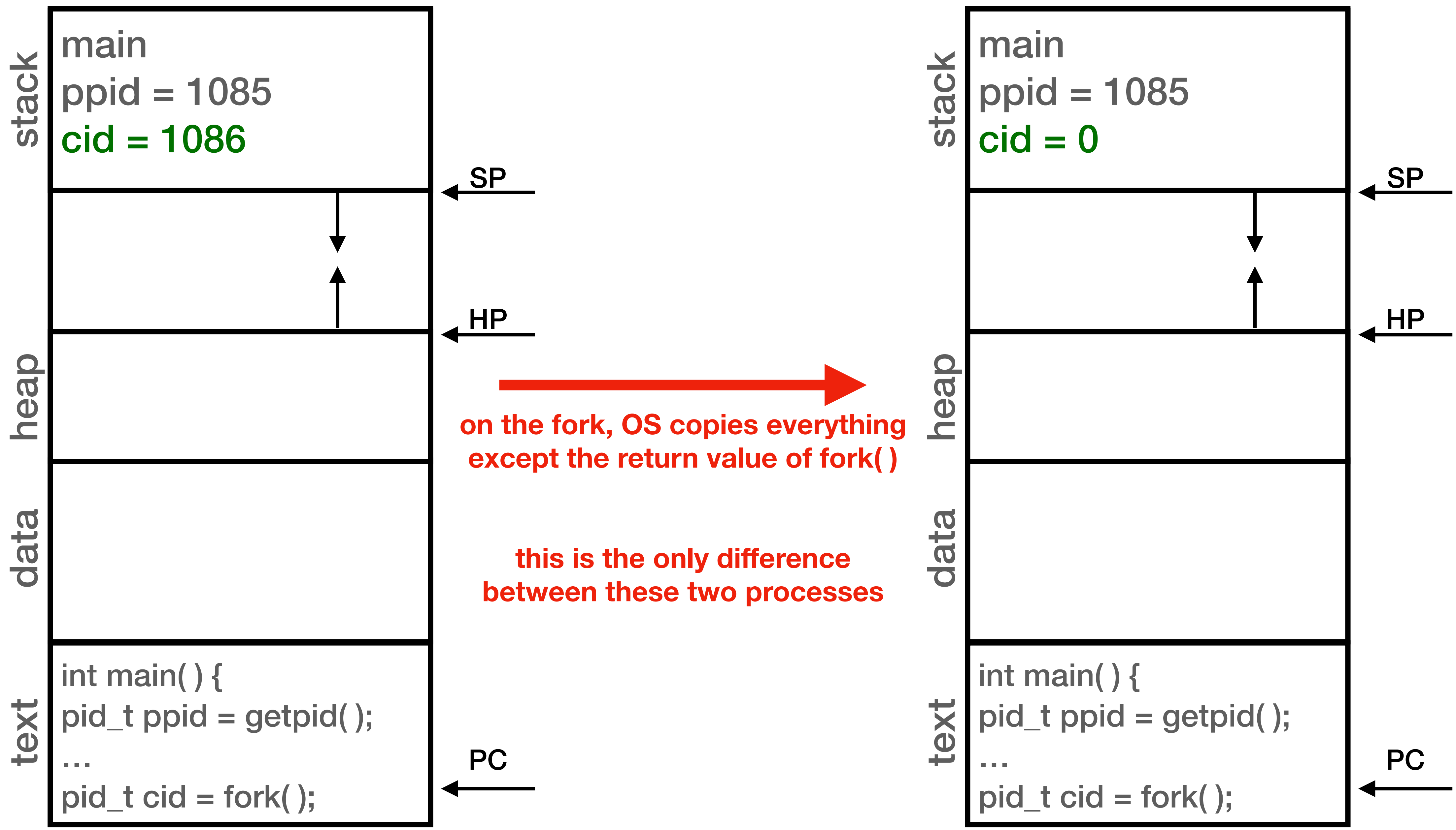
- The `fork( )` system call creates a **child** process that inherits a copy of its **parent**'s memory, open file descriptors, CPU registers, etc.
- Both parent and child processes execute from the instruction following `fork( )`
  - does the child or parent process run first? We don't know!
- The return value from `fork( )` is of type pid_t (like an integer)
  - when > 0: running in (original) parent process and the return value is pid of new child
  - when = 0: running in new child process
  - when < 0: error! must handle somehow still running in original process



the `systemd` process is the root of this tree (parent of all user processes created when the system has booted); it has PID =1

What happens on a fork?

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



Common problems with fork

The `fork( )` system call is

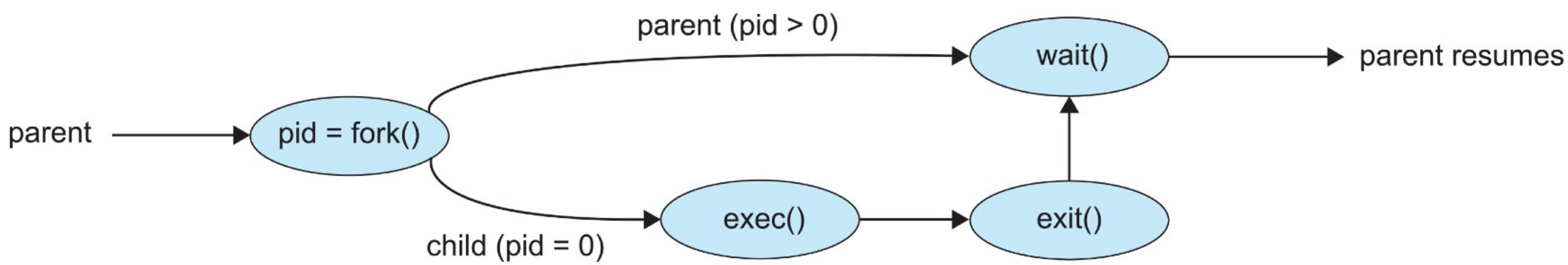
- inefficient and slow
  - the cost of copying the entire address space of a process is high
- insecure
  - the parent process must explicitly remove states that the child process does not need (scrubbing secrets from memory)
- not thread-safe
  - the child process created by the fork system call will have a single thread only (a copy of the calling thread)
  - **Problem**: one thread doing memory allocation and holding a **heap lock**, while another thread forks. Any attempt to allocate memory in the child (and thus acquire the same lock) will immediately **deadlock** waiting for an unlock operation that will never happen
  - **Solution**: not using fork in a multithreaded process, or calling exec immediately afterwards

Program loading with the exec system call

- The `execve( )` system call allows a process to load a different program and start execution from its main function
  - allows a process to specify the number of arguments (argc) and the string argument array (argv) that must be sent to the new process
- If the call is successful, the same process runs a different program
  - code, data, stack and heap sections are overwritten
- We normally call `execve( )` right after calling `fork( )`
  - hence, the memory copied during `fork( )` is useless
  - in performance-sensitive applications, the `vfork( )` system call allows creating a process without creating an identical memory image; in this case child process must call `execve( )` immediately — undefined behaviour if it doesn't

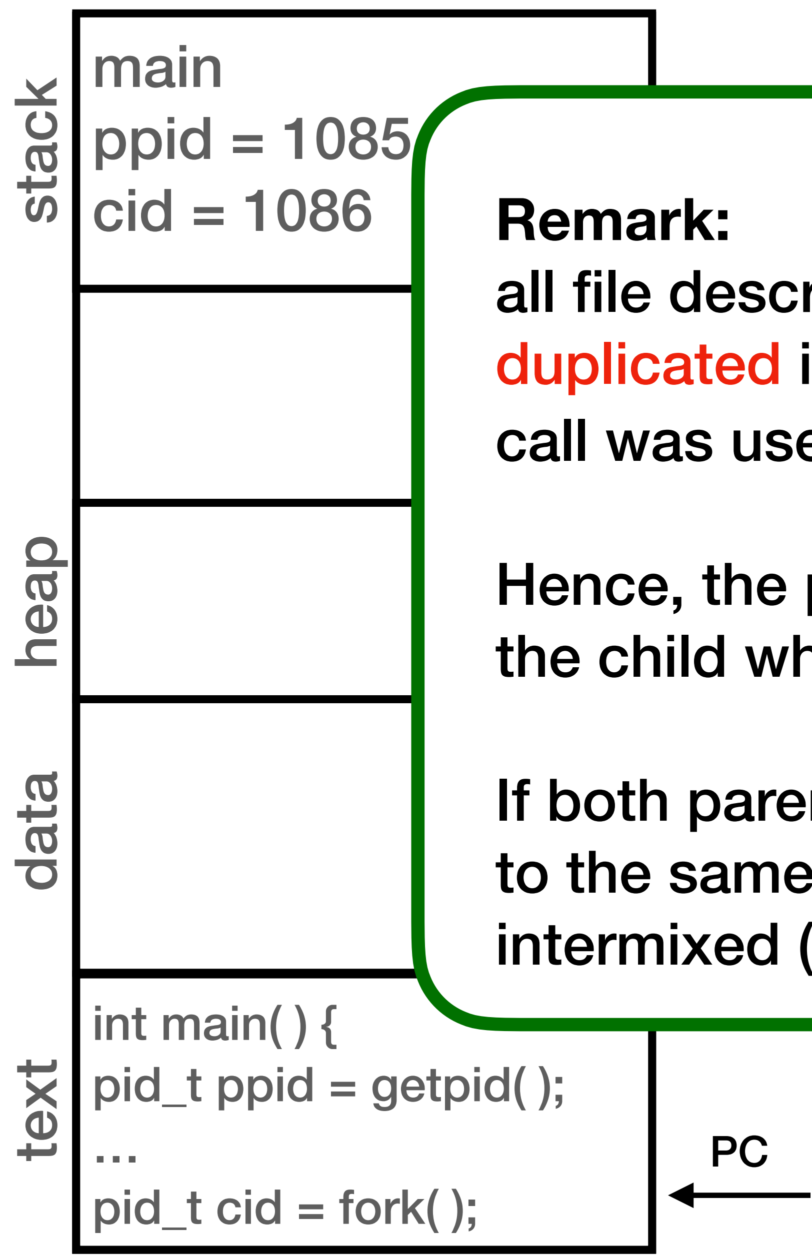
Waiting for the child process to terminate

- 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 a child process to change state (e.g. terminate)
  - 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 return 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 return values immediately (and deallocates the zombie)



What happens on a fork?

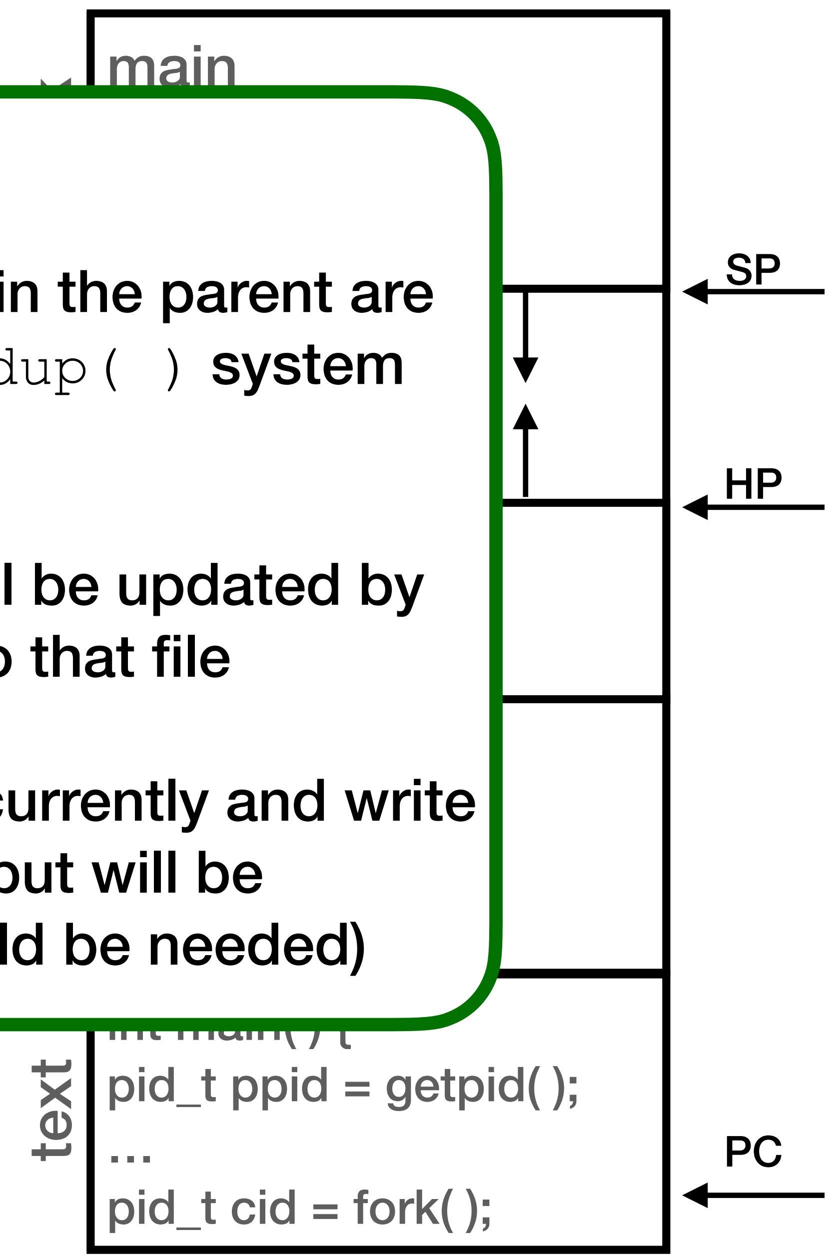
```
pid_t cid = fork( );
```



Remark:
all file descriptors that are open in the parent are duplicated in the child (as if the `dup( )` system call was used)

Hence, the parent's file offset will be updated by the child when the child writes to that file

If both parent and child run concurrently and write to the same descriptor, their output will be intermixed (synchronization would be needed)



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 data 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( )`;
  - open file descriptors are closed; children are inherited by the `init` process
- When the `main` function returns, `exit( )` is called indirectly
  - `exit( )` calls all exit handlers that have been registered using `atexit( )` — a glibc function
  - `_exit( )` does not call exit handlers
- For abnormal termination of a process, 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)`

Zombie and orphan processes

- A process that has terminated, but its parent has not (yet) called `wait( )` becomes a 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
         printf("Child pid: [%d]\n", getpid());
         ...
      } else if (pid > 0) {                   // parent continues
        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
        ...
      } else if (pid > 0) {                       // parent continues
        …
        pid_t cpid = wait(&child_status);         // how was it stopped or terminated
      } 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
         execve("/bin/ls", arg0, arg1, …); // mark the end with a null pointer
          // exec doesn't return on success! so if we got here, it must have failed!
         perror("exec failed!");
      } else if (pid > 0) {                     // parent continues
         ...
         cpid = wait(&status);                  // pass NULL if not interested in exit status
         if (WIFEXITED(status))                 // true if the child terminated normally
             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:
  - 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 by having its system calls intercepted; very useful for breakpoint debugging
    - 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` **system call** sends a signal to a process or process group based on the specified PID
- the `kill` **command** 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
    - `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

- Acts as a process control system
  - allowing programmers to create and manage a set of processes to do some tasks
  - Windows, Linux, MacOS have their own shells
- When you log in to a machine running UNIX, you create a shell process
- Every command launched in the shell is a child process of the shell process (an implicit `fork( )` and `execve( )` pair)
- The separation of `fork( )` and `execve( )` enables features like input/output redirection, pipes, etc.
  - the shell runs code after the call to `fork( )` and before the call to `execve( )`

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