The Process API

ICS332
Operating Systems

Henri Casanova (henric@hawaii.edu)

Disclaimer

- Most of the content of this set of lecture notes is for UNIX-like OSes
  - I won’t have “in UNIX-like OSes” on very slide
- There will be a bit of content about Windows though

Process Creation

- Processes can create processes
- If process A creates process B, we say that “A is the parent of B” and “B is a child of A”
  - A process can have at most one parent and can have many children
- Each process has a PID (Process ID)
  - An integer picked by the OS, always increasing
  - If I just created a process and its PID is 456, then the next process that will be created (by any one) will have PID 457
  - Therefore, if I just created a process and it’s PID is 1000, I know that 1000 processes have been created since booting the machine (most of which have died since, and assuming that the first one had PID 1)
- The PID of the parent of a process is called the PPID (Parent Process ID)
- Two useful system calls: **`getpid()`** and **`getppid()`**
- Bottom line: Processes form a genealogy tree!

Looking at the Process Tree

- On Mac OSX: ps axlw

```
UID   PID  PPID CPU PRI NI      VSZ    RSS WCHAN  STAT   TT       TIME COMMAND
[...]
501  2660     1   0  31  0  2458784    536 -      Ss     ??    0:00.19 gpg-agent --daemon
501  2667     1   0  31  0  2467676    676 -      S      ??    0:00.00 /opt/X11/libexec/launchd_startx /opt/X11/bin/
501  2668  2667   0  31  0  2439512   1064 -      S      ??    0:00.01 /bin/sh /opt/X11/bin/startx -- /opt/X11/bin/X
501  2733  2668   0  31  0  2452676    836 -      S      ??    0:00.00 /opt/X11/bin/xinit /opt/X11/lib/X11/xinit/xin
501  2734  2733   0  31  0  2479128   2704 -      S      ??    0:00.01 /opt/X11/bin/Xquartz :0 -nolisten tcp -iglx -
501  2736  2734   0  63  0  2654600  46768 -      S      ??    0:06.31 /Applications/Utilities/XQuartz.app/Contents/
501  2743     1   0  31  0  2450592    532 -      Ss     ??    0:00.19 gpg-agent --daemon
501  2836  2733   0  31  0  2550224   7108 -      S      ??    0:00.07 /opt/X11/bin/quartz-wm
[...]
```

- On Linux: ps --forest -eaf

```
UID        PID  PPID  C STIME TTY          TIME CMD
[...]
daemon    1061     1  0 Aug04 ?        00:00:00 /usr/sbin/atd -f
root      1063     1  0 Aug04 ?        00:00:00 /usr/bin/lxcfs /var/lib/lxcfs/
syslog    1069     1  0 Aug04 ?        00:00:00 /usr/sbin/rsyslogd -n
root      1074     1  0 Aug04 ?        00:00:00 /usr/sbin/sshd -D
root     25393  1074  0 01:31 ?        00:00:00  \_ sshd: ubuntu [priv]
ubuntu   25453 25393  0 01:31 ?        00:00:00      \_ sshd: ubuntu@pts/0
ubuntu   25454 25453  0 01:31 pts/0    00:00:00          \_ -bash
ubuntu   25509 25454  0 01:35 pts/0    00:00:00              \_ ps --forest -eaf
root      1081     1  0 Aug04 ?        00:00:01 /usr/lib/snapd/snapd
root      1118     1  0 Aug04 ?        00:00:00 /sbin/mdadm --monitor --pid-file /run/mdadm/monitor.pid --daemoni
[...]
```

The pstree program

- On ubuntu, the **`psmisc`** package comes with a cool program called **`pstree`**
- Let's go to my Linux box and play with it
- For instance: **`pstree -c -C age -G -T`**

Process Creation

- After creating a child the parent continues executing
- But at any point, event right away, it can wait for the child's completion
- The child can be:
  - either a complete clone of the parent (i.e., have an exact copy of the parent's address space)
  - or be an entirely new program
- The above is true across most modern OSes, more or less, but comes with important variations
- Let's look at process creation in the POSIX standard
  - UNIX (mostly Linux these days)
  - Darwin (MacOS + iOS + tvOS + watchOS)

- Let's begin with the strange and powerful fork()

The fork() System Call

- `fork()` is a system call that creates a new process
  - It's really a thin wrapper over the `clone()` system call
  - But `fork()` is kept as a system call for backward compatibility reasons
- The child is an almost exact copy of the parent except for
  - Its PID (two processed cannot have the same PID)
  - Its PPID (its parent cannot also be its grand-parent)
  - Its resource utilization (set to zero since it's just started)
- After the call to `fork()` the parent continues executing and the child begins executing
- **The confusing part:** `fork()` returns an integer value
  - **It returns 0 to the child**
  - **If returns the child's PID to the parent**
  - (In case of error, e.g., the Process Table is full, it returns -1)

fork(): Basic Example

The basic use of fork()

```
returnedValue = fork();
if (returnedValue < 0) {
  // Manage the error
   printf("Error: Can't fork!\n");
} else if (returnedValue == 0) {
  // Child code
   printf("I am the child and my pid is %ld\n", getpid());
   while (1==1); // I just don't want to terminate
} else {
  // Parent code
  print("I am the parent and the pid of my child is %ld\n", returnedValue);
  while (1==1); // I just don't want to terminate either
}
```

- Simplified version of fork_example1.c
- Note: Errors cases should always be handled… but perhaps doing so for **`printf`** is overkill :)
- Let's run it…

fork(): Second Example

Second example of fork()

```
a = 12; // Global variable
pid_t pid = fork();
if (pid) {
   // The PARENT
   sleep(5); // Ask the OS to put me in the WAITING state for 5s
   printf("a = %d", a); // Display the value of a
   while (1); // Loop forever
} else {
   // The CHILD
   a += 3;
   while (1); // Loop forever
}
```

- What does this code print? 12 or 15?

fork(): Second Example

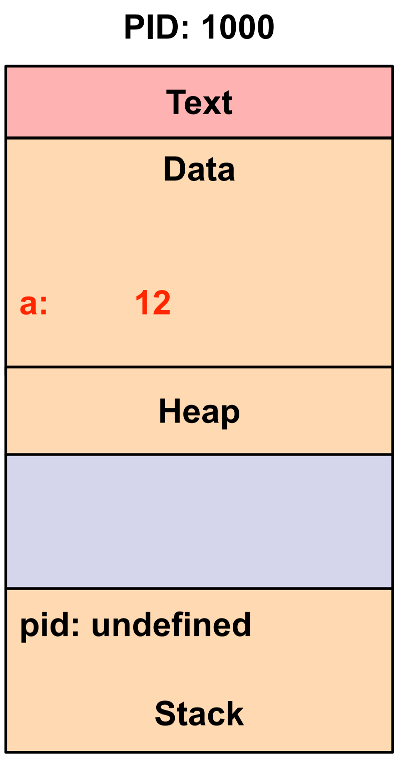
Second example of fork()

```
a = 12; // Global variable
pid_t pid = fork();
if (pid) {
   // The PARENT
   sleep(5); // Ask the OS to put me in the WAITING state for 5s
   printf("a = %d", a); // Display the value of a
   while (1); // Loop forever
} else {
   // The CHILD
   a += 3;
   while (1); // Loop forever
}
```

- What does this code print? 12 or 15?
- **It prints 12** fork_example_2.c
- Let's look at this in full detail…

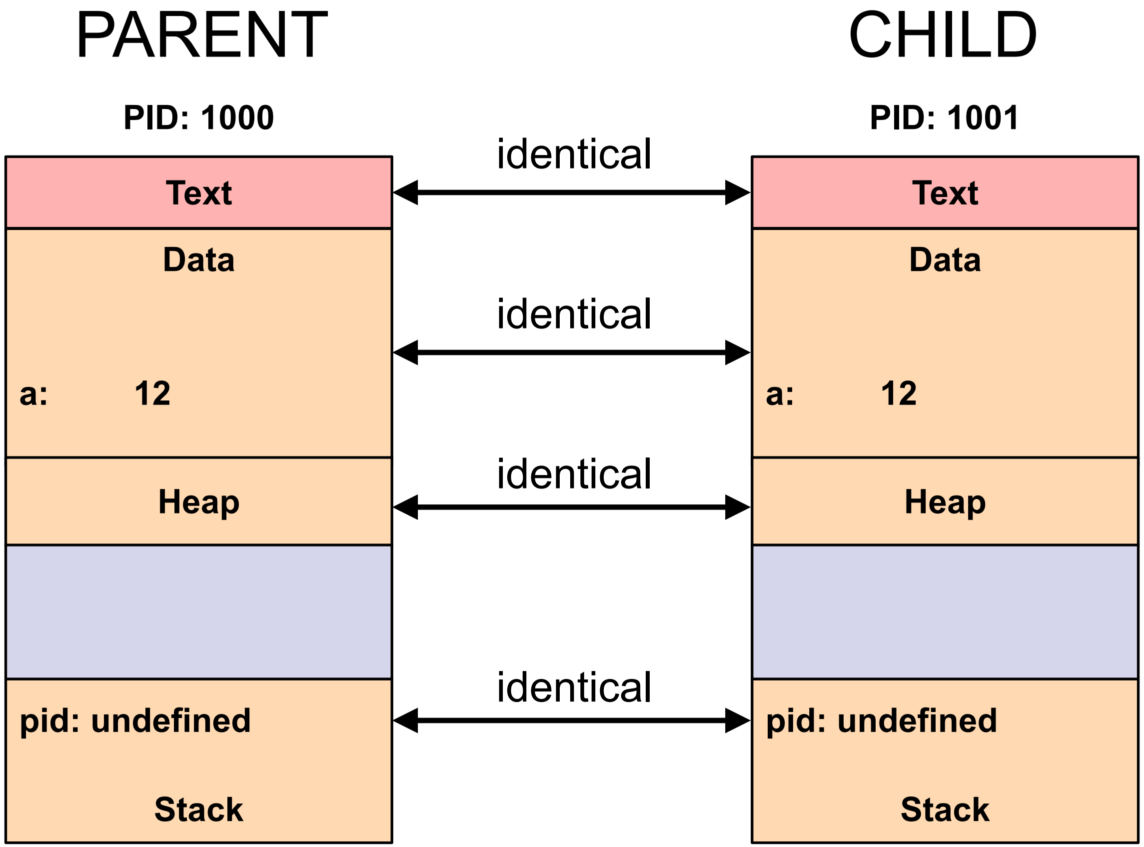
fork(): Second Example

```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



fork(): Second Example

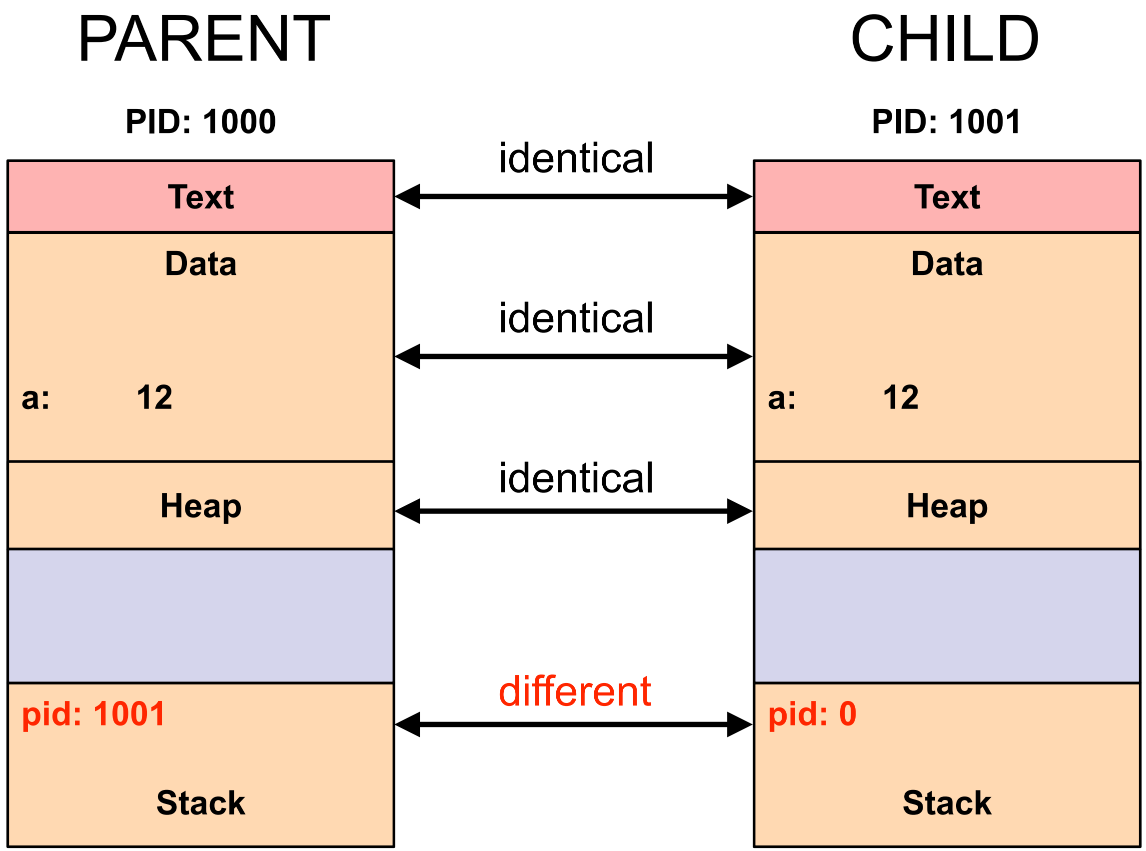
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



Right after **`fork()`** and before the assignment to **`pid`**

fork(): Second Example

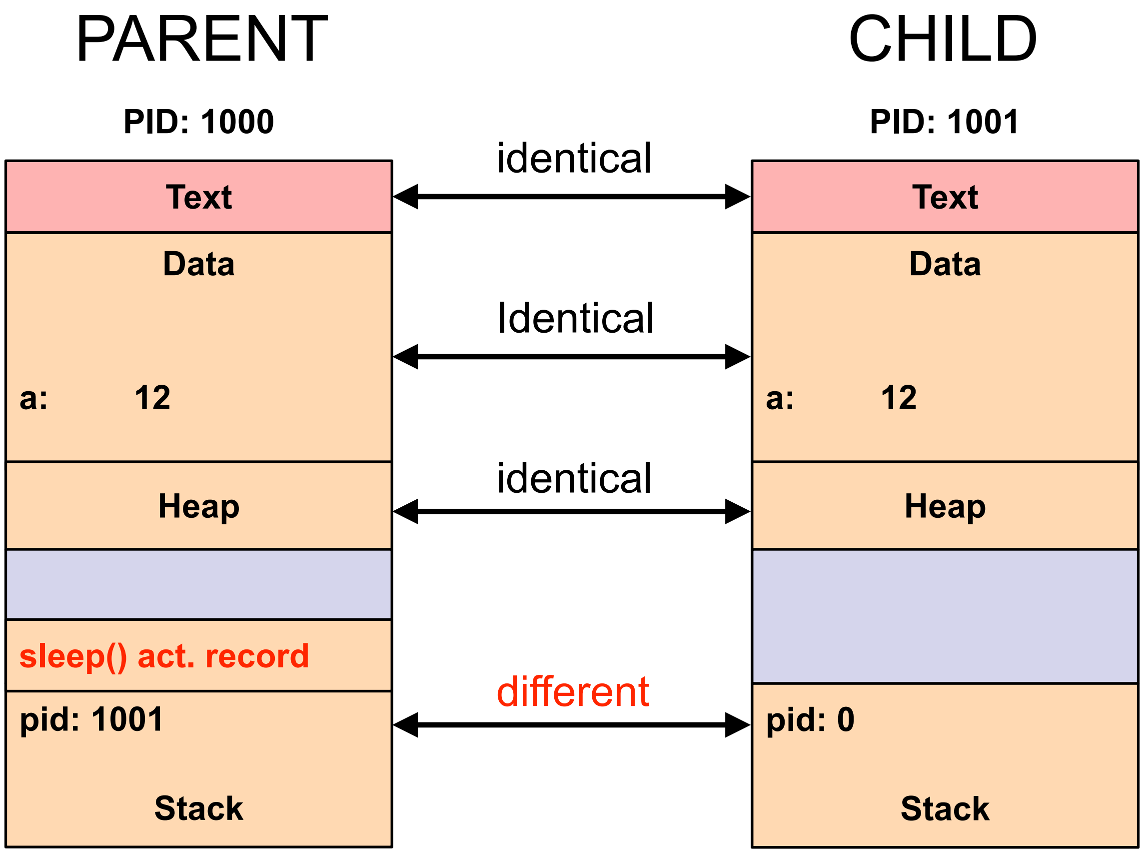
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



After the assignment to `pid`

fork(): Second Example

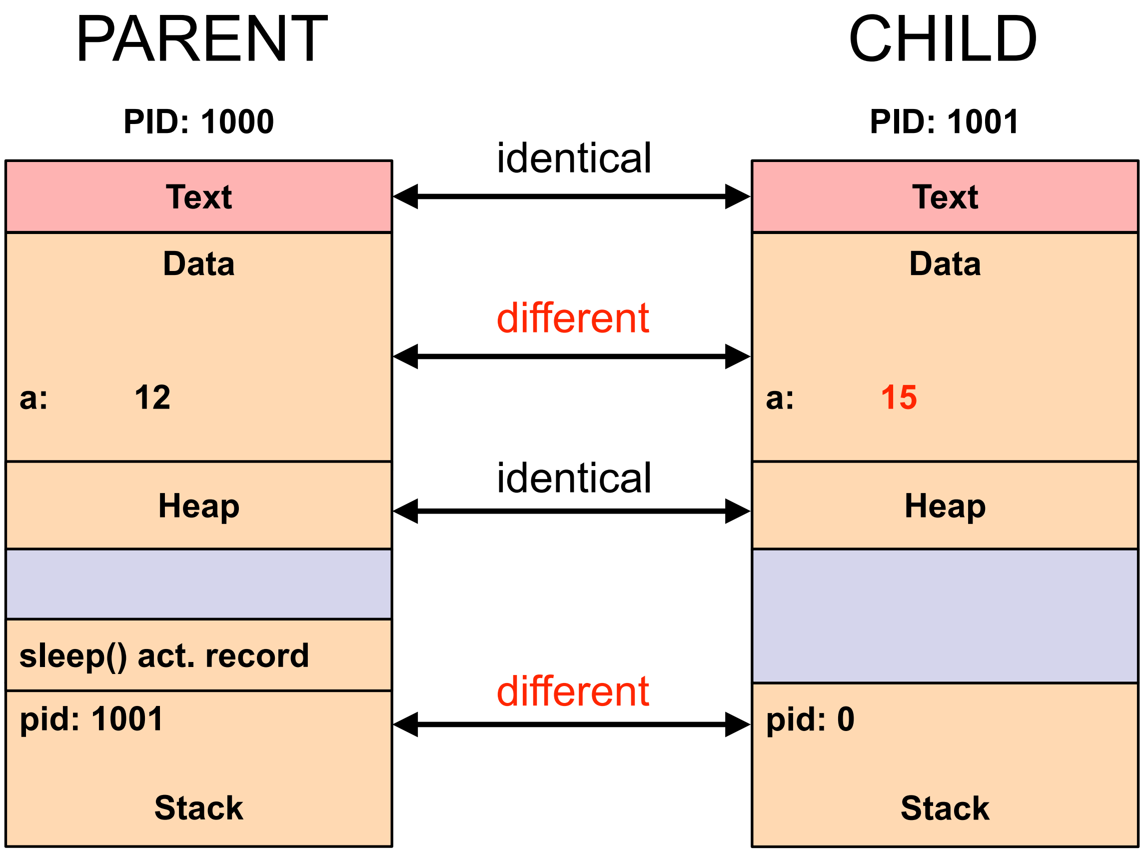
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



The parent calls **`sleep()`**, goes to the waiting state, which will let the child run

fork(): Second Example

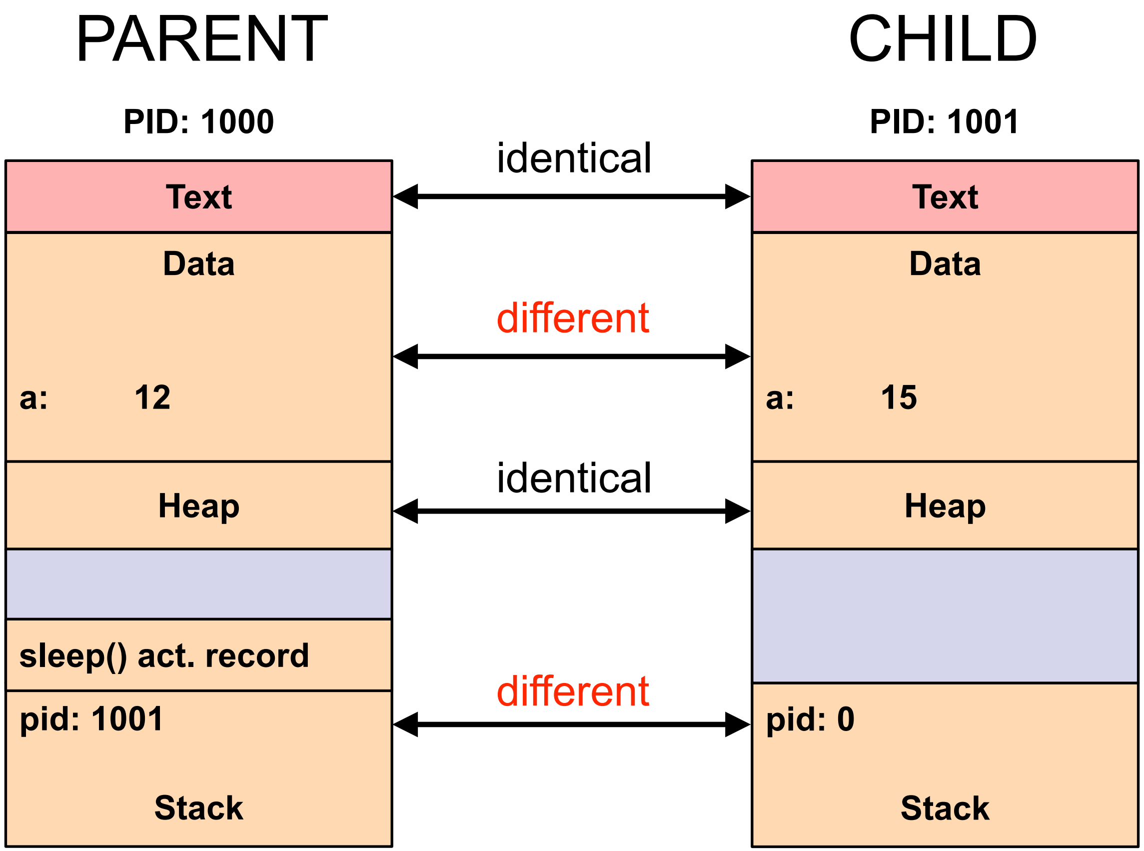
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



The child runs, and updates **its** values of `a` to 15

fork(): Second Example

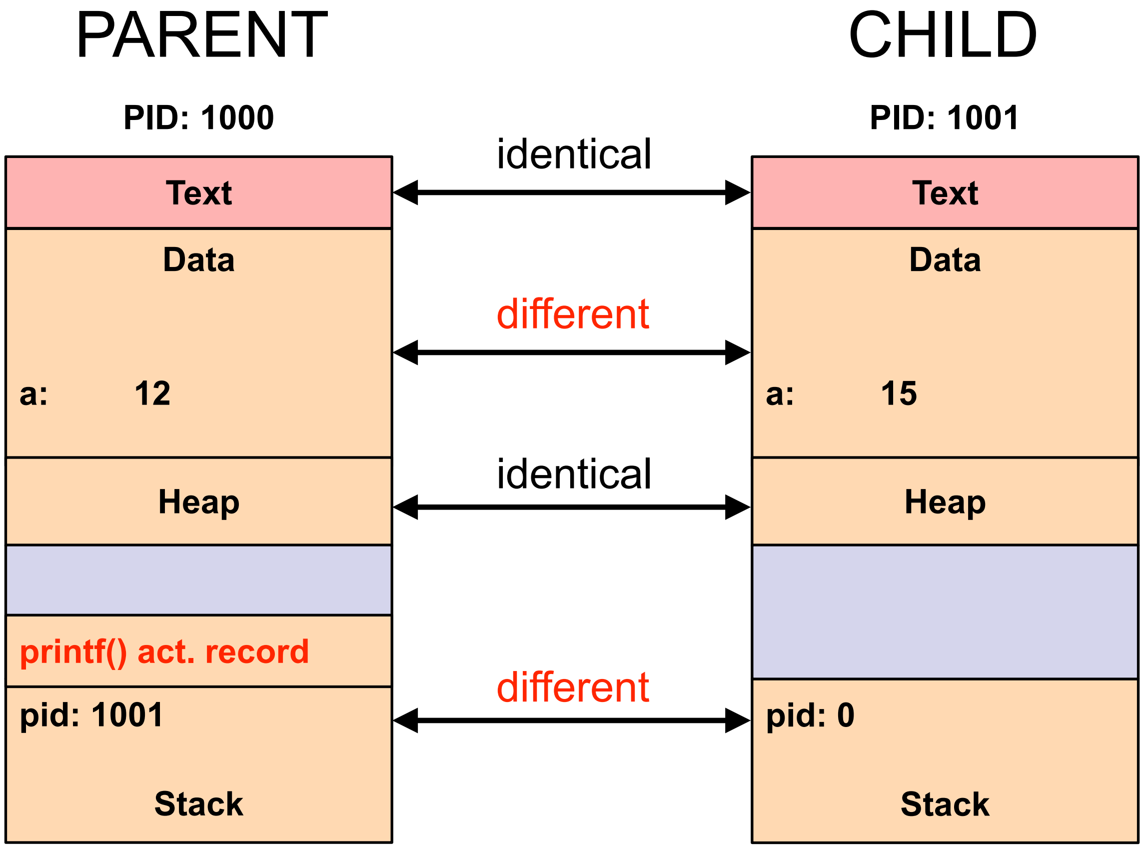
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



The child does an infinite loop, and at some point will be interrupted so that another process gets to run

fork(): Second Example

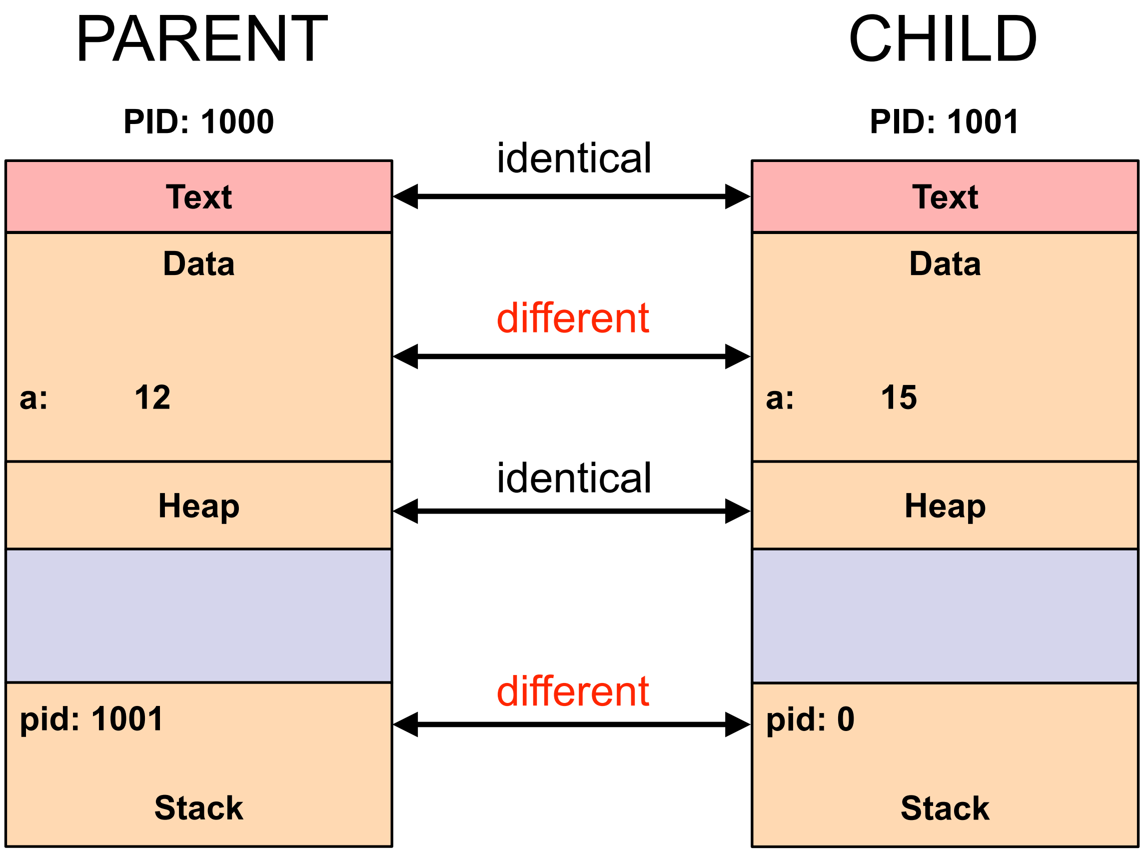
```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



The parent calls **`printf()`** and prints 12 (**its** value of a)

fork(): Second Example

```
a = 12;
pid_t pid = fork();
if (pid) {
   sleep(5);
   printf("a = %d", a);
   while (1);
} else {
   a += 3;
   while (1);
}
```



`printf()` returns and the parent goes into its own infinite loop

Second Example's Lesson

- Both processes coexist independently
  - The code is executed independently in the Parent and in the Child
  - The data segment of the Parent has **nothing to do** with the data segment of the Child
  - The stack of the Parent has **nothing to do** with the data segment of the Child
  - The heap of the Parent has **nothing to do** with the data segment of the Child
  - This is by design, because the OS ensures that each process has its own address space!

- Let's look at a small variation of the example and see if we can figure it out…

fork(): Second Example, Tweaked

Second example of fork(), tweaked

```
int a = 12;
retVal = fork();
if (retVal) {
   // The PARENT (or error)
   sleep(5); // Ask the OS to put me in the WAITING state for 5s
} else {
   // The CHILD
   a += 3;
}

printf("%d\n", a); // Display the value of a
```

- What does this code print?

fork(): Second Example, Tweaked

Second example of fork(), tweaked

```
int a = 12;
retVal = fork();
if (retVal) {
   // The PARENT (or error)
   sleep(5); // Ask the OS to put me in the WAITING state for 5s
} else {
   // The CHILD
   a += 3;
}

printf("%d\n", a); // Display the value of a
```

- What does this code print?
- **It prints 15\n12\n** fork_example3.c

fork() is sometimes confusing

fork() and printing "Hello"

```
fork();
printf("Hello");
fork();
print("Hello");
```

- How many times does this program print Hello? (Show of hands)

fork() is sometimes confusing

fork() and printing "Hello"

```
fork();
printf("Hello");
fork();
print("Hello");
```

- How many times does this program print Hello? (Show of hands)
- Answer: 6 times   fork_example4.cx
  - One process calls fork()
  - Two processes print "Hello"
  - Two processes call fork()
  - Four processes print "Hello"

fork(): A crazy example

fork() gone crazy

```
fork();
if (fork()) {
  fork();
}
fork();
```

- How many processes does this C program create?
  - Note the typical C coding style for condition in the conditional (true if fork() returns non-zero)
- Let’s go through this together in the next slides…
  - Clearly the above program is not useful
  - But if you can figure it out, that means you understand fork() 100%

fork(): A crazy example

Initial process

```
fork();
if (fork()) {
  fork();
}
fork();
```

fork(): A crazy example

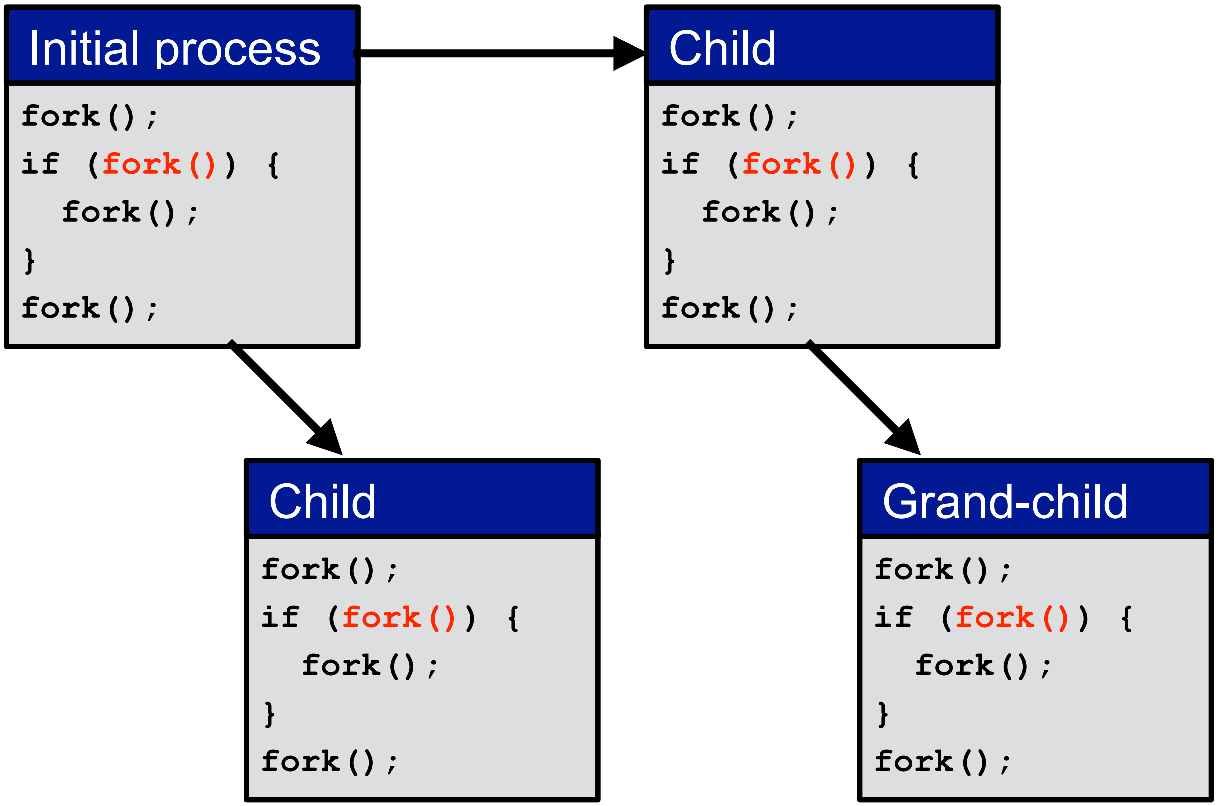
Initial process

```
fork();
if (fork()) {
  fork();
}
fork();
```

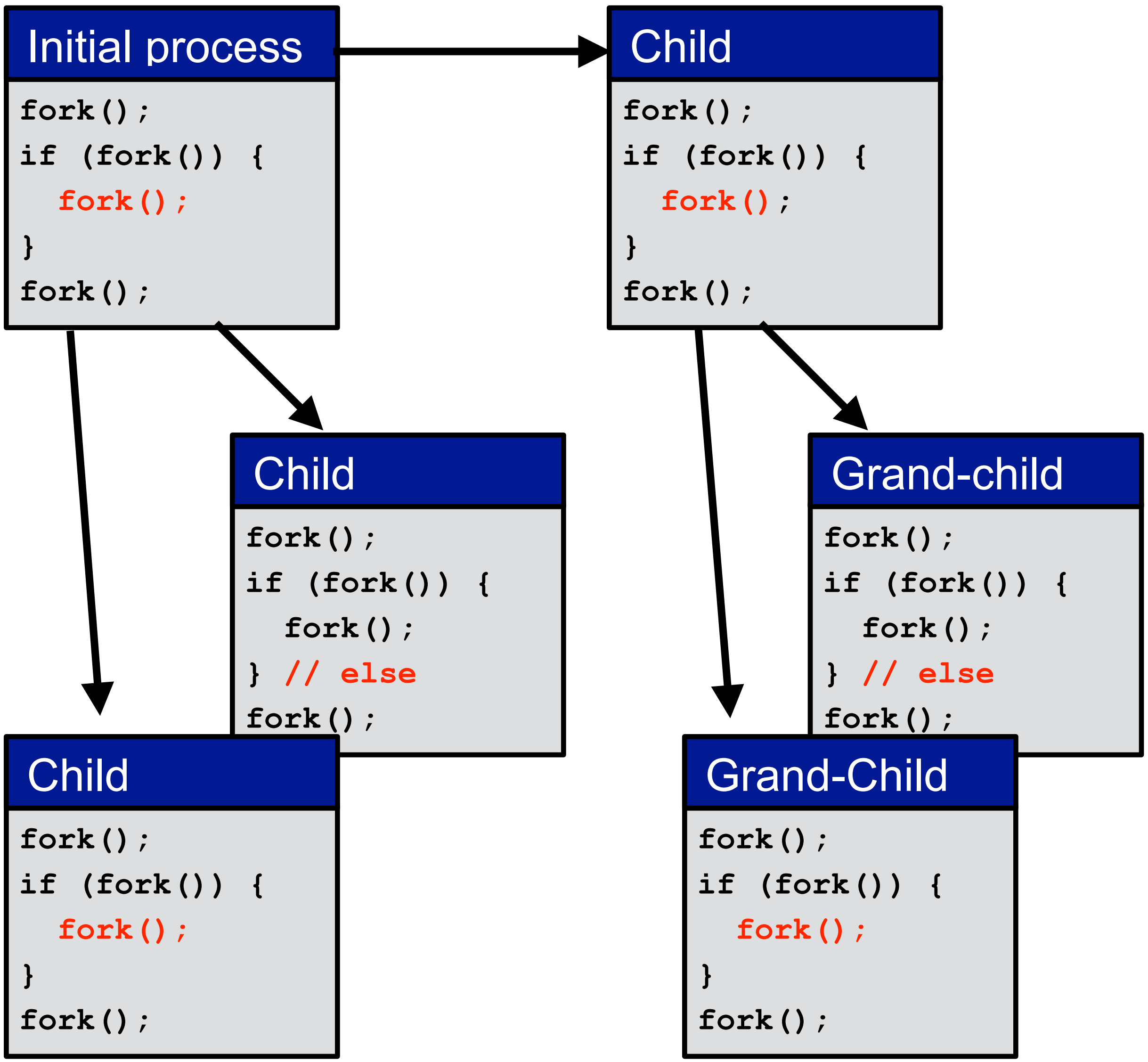
Child

```
fork();
if (fork()) {
  fork();
}
fork();
```

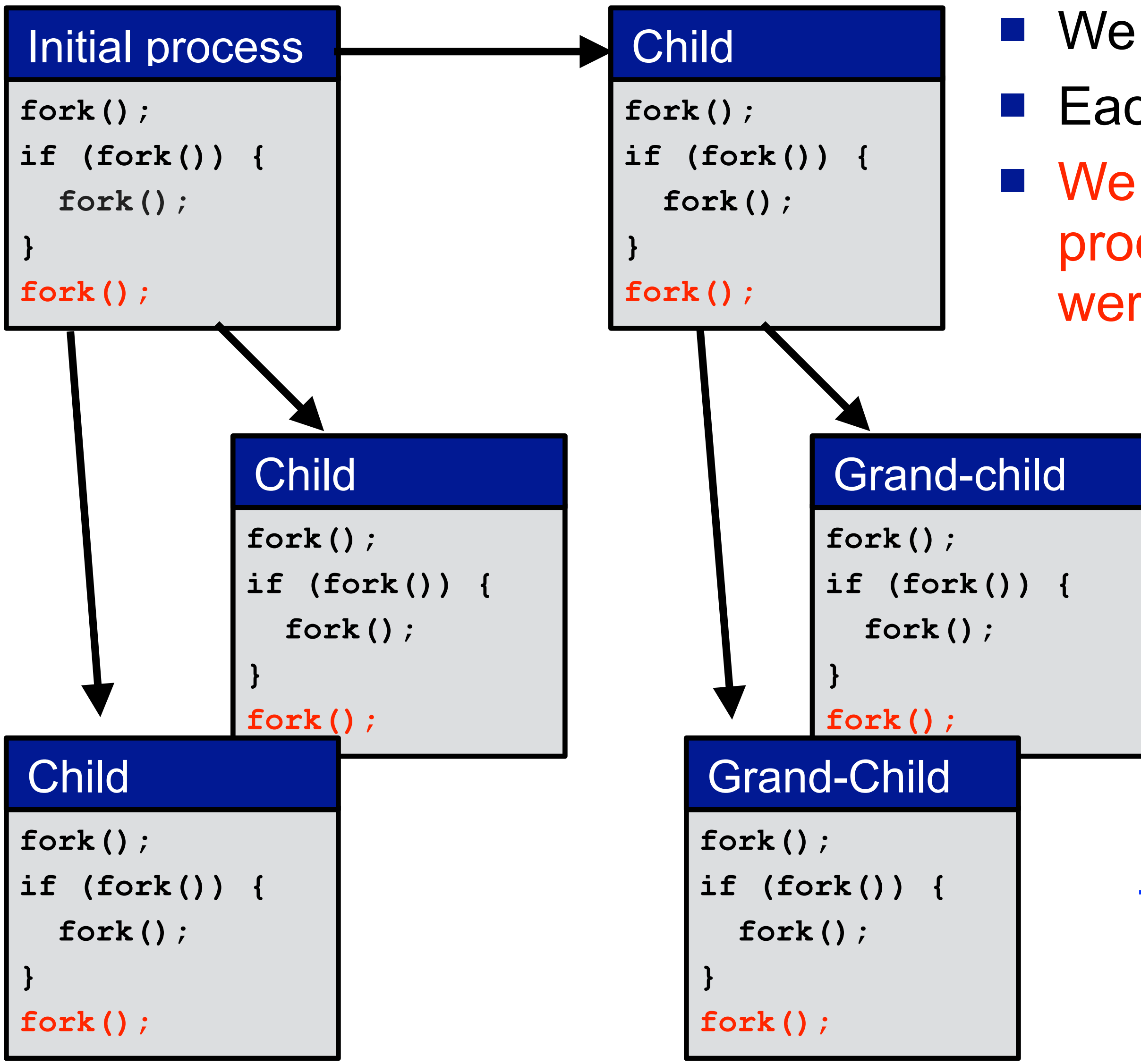
fork(): A crazy example



fork(): A crazy example



fork(): A crazy example



- We now have 6 processes
- Each calls fork()
- We end up with 12 processes in total (11 were created)

fork_example5.c

Filling up the Process Table

A fork bomb!

```
for (;;) {
  fork();
}
```

- The above program will fill up the process table
- This is often called a "fork bomb", and is typically a bug (I've seen it happen more than once!)
- The result is that the system becomes unusable and has to be hard-rebooted
- Typically the OS will bound the number of processes a user can create
- One can change that limit: **`ulimit -u <count>`**
  - And one can check on what that limit is: **`ulimit -u`**
- But as a user, if you reach that limit, although you won't take down the system, you won't be able to use it at all…

fork() clones more than you think

fork() and buffered I/O

```
printf("BUF");      // "BUF" is not flushed to the terminal
                    // but stored in a buffer (no newline character!)
fork();             // Will clone that buffer!
printf("FER\n");    // BOTH processes print "BUFFER" to the terminal!
```

- Terminal output is always **bufferized**: nothing gets printed to the terminal unless:
  - A newline character is output
  - The program explicitly calls `flush()`
  - The program terminates
- Most language implementations do this to boost performance:
  - Copying data to a buffer is very fast
  - Displaying data on the terminal involves a system call, which has much higher overhead
- This is also done for writing to the disk (hard drive, SSD) since that has huge overhead compared to just copying data to a buffer
  - This is why, if you're in the middle of writing data to disk and the machine abruptly crashes some data is likely lost
  - It was in a buffer and was waiting to be "flushed" to the disk!

The `exec*` Syscall Family

- **`man 3 exec: execl, execlp, execle, execv, execvp, execvpe`**
- These are all variations of the "exec" syscall: replaces the process image (i.e., the process' address space) by that of a specific program (stored on disk as an executable)
- You give exec:
  - A path to an executable
  - A list of command-line arguments for that executable
  - A set of environment variables
- **The call to exec never returns** unless there is an error, and your running program is now another running program

Exec: Basic Example

Basic exec example

```
int main(int argc, char *argv[]) {
  char* const args[] = {"ls", "-l", "/tmp", NULL};
  execv("/bin/ls", args);
  printf("This never gets executed...\n");
}
```

- The above program immediately “becomes” the `ls` program invoked with arguments `-l /tmp`
- exec_example1.c

Exec: Combined with fork()

The quintessential fork-exec example

```
if (fork() == 0) {
    // Child
    char* const args[] = {"ls", "-l", “/tmp", NULL};
    execv("bin/ls", args);
} else {
    // Parent
    for (;;);
}
```

- This is exactly how the Shell is able to run commands!
- exec_example2.c

The Living Dead???

- Let’s run the program on the previous slide on Linux and look at the running processes…

```
PID TTY         STAT    TIME COMMAND
  1 pts/0       Ssl     0:00 /bin/bash
 29 pts/0       Rl      0:05 ./exec_example4
 32 pts/0       Z       0:00  \_ [ls] <defunct>
```

The Living Dead???

- Let's run the program on the previous slide on Linux and look at the running processes…

```
PID TTY          STAT   TIME COMMAND
  1 pts/0        Ssl    0:00 /bin/bash
 29 pts/0        Rl     0:05 ./exec_example4
 32 pts/0        Z      0:00  \_ [ls] <defunct>
```

- Defunct (from the Latin defunctus) means dead
- The "Z" stands for Zombie

Zombie Processes

- When a child process dies, it remains as a zombie in the Terminated state
  - Recall that in the Process Lifecycle diagram, we had a Terminated state, which some of you might have thought a bit useless?
- Why??? The parent process may want to know about the status of a child that has died in the past to see what happened to it
  - We'll see how to do that in a bit
- The OS keeps zombies around for this purpose:
  - Zombies do not use hardware resources, but a slot in the Process Table!
  - The Process Table may fill up due to Zombies (and cause **`fork()`** to fail)
- A zombie lingers until
  - Its parent has acknowledged its death, or
  - Its parent dies
- The zombie is then "reaped" by the OS
- It is very frowned upon to leave zombies around unnecessarily

- And yes, this is all very dark/macabre…

Process Termination

- To understand how to get rid of zombies, we need to learn a bit more about process termination
- A process terminates itself with the `exit()` system call, which takes as argument an integer called the process exit|return|error value|code
- All resources of the process are then deallocated by the OS (memory, open files, I/O buffers, …)
  - But the PCB main remain in the Process Table as a zombie
- A process can also cause the termination of another process
- This is done using signals and the `kill()` system call...

Signals

- Signals are software interrupts, i.e., a signal is an asynchronous event that a program must act upon in some way
- The OS defines a number of signals, each with a name and a number, and some "default" meaning
  - See man 7 signal
- Signals happen for various reasons:
  - ^C on the command-line sends a SIGINT ("Interrupt from keyboard") signal to the running program in the Shell
  - Invalid access to valid memory sends a SIGSEGV signal to the running process (e.g., trying to write to read-only memory)
  - Tying to access an invalid address sends a SIGBUS signal to the running process (e.g., trying to de-reference and non-allocated pointer)
  - A process can send a SIGKILL signal to another process to kill it
- Signals can be used for process synchronization ("hey! do something!"), but we'll see other more powerful/flexible synchronization mechanisms

Signal Handlers

- Each signal causes a default behavior in the process
  - e.g., the **`SIGINT`** signal causes the process to terminate
- The **`signal()`** syscall allows a process to specify what to do when a signal is received
  - **`signal(SIGINT, SIG_IGN);    // Ignore SIGINT`**
  - **`signal(SIGINT, SIG_DFL);    // Default behavior`**
  - **`signal(SIGINT, my_handler);// Custom behavior`**
- Let's look at **`signal_example.c`**
- Some signals cannot be reprogrammed by the user: **`SIGKILL`**, **`SIGSTOP`**, etc.

Back to Zombies: `wait()` and `waitpid()`

- A parent can wait for a child's completion
- The **`wait()`** syscall – See wait_example1.c
  - Blocks until any child completes
  - Returns the pid of the completed child and the child's exit code
- The **`waitpid()`** syscall
  - Blocks until a specific child completes — See wait_example2.c
  - Can be made non-blocking — See wait_example3.c
- One way to avoid zombies: always call **`wait()`** or **`waitpid()`**
- This seems easy enough, but sometimes really inconvenient
  - e.g., I am a Web server, and each time I get a request for some content I spawn a process to handle it
  - The Web server really doesn't need to "wait" for children processes to terminate; it wants to "fork and forget"
  - The only goal of waiting would be to avoid zombies... how annoying!
- So how do we do this?

The SIGCHLD signal

- When a child exits, a SIGCHLD signal is sent to the parent
  - This is implemented by the kernel
- The typical convenient way to avoid zombies altogether:
  - The parent associates a handler to SIGCHLD
  - The handler calls **`wait()`**
  - This way all children terminations are acknowledged
- See wait_example4.c
- We can now write zombie-free code:
  - If you need to wait for a child process to terminate, then great, call **`wait()`**
  - And create a handler that will asynchronously call **`wait()`** for those children you don't want to explicitly wait on
  - This way, **`wait()`** is called for all children

Orphans

- What happens when a parent dies before its child?
- The child becomes an orphan
- Let's run orphan_example1.c
  - We see that the child keeps running even after its parent has terminated!
- Who becomes responsible for the orphan?
- Let's run orphan_example2.c in which the child prints its PPID
- The orphan has been adopted by the process with PID 1
  - On Linux this used to be the famous `/sbin/init` program (on recent Linux, the adopter is `/lib/systemd/systemd`)
  - On MacOS this is the `/sbin/launchd` process
- Having orphan processes could be a bug or a feature of your code

Giving Up Parental Responsibilities

- To create a child process that is completely separate from the parent: create a grandchild and kill its parent (I know, it's *horrible*)

Bad grandpa

```
if (!fork()) { // Child
  if (!fork()) { //Grandchild
    ...
    exit(0); //Will be orphaned and then reaped by init
  }
  exit(0) //Will be reaped by bad grandpa
} else {
  // Grandpa
  wait(NULL); // Wait for the child to exit, so that it's not zombified
}
// At this point, I am the Grandpa and I have no responsibilities,
// because my grandchild has been adopted by PID 1
```

- The process with PID 1 has adopted the grandchild
- It is responsible and calls **`wait()`** is a handler, so the grandchild will not become a zombie
- Useful to start a process and logout
  - The **`screen`** command does this and is a life-saver for the command-line user!

Is all of this useful?

- It’s hard to see it is, until it saves your (developer) life
- I am currently working on an open-source project, in which we’ve used fork/exec in various ways
- Use #1: We use the Google Test framework
  - Google Test does not perform each test in its own process
  - So if one test totally crashes, then the tests abort
    - Yes, a test shouldn’t crash, but reality bites
  - Solution: simple, create a child process using **`fork()`**
- Use #2: We need to start a daemon as a separate process, and we need it to die if our main process dies
  - We use fork-exec to start the process
  - We do clever fork/exec/pipe magic to have that process die if we die
  - Let’s look at the code…

What about Windows?

- The Windows documentation is clear: "*One of the largest areas of difference [in porting UNIX applications to Windows] is in the process model. UNIX has fork; Win32 does not.*"
- In Windows, the **`CreateProcess()`** call combines **`fork()`** and **`exec()`**
  - Separation of fork and exec allows many clever "tricks" in UNIX, which are not possible in Windows
  - From [The Evolution of the Unix Time-sharing System](): "In PDP-7 fork() required precisely 27 lines of assembly code" ... "a combined fork-exec [`a la Windows] would have been considerably more complicated"
- There is an equivalent to **`wait()`**: **`WaitForSingleObject()`**
- There is an equivalent to **`kill()`**: **`TerminateProcess()`**
- So, overall, Windows allows for the same capabilities as UNIX (which shouldn't be surprising), but with a different flavor

Main Takeaways

- The **`fork()`** system call
- The **`exec*()`** system call(s)
- The **`wait()`** and **`waitpid()`** system calls
- Orphans and Zombies
- Signals and how the **`SIGCHLD`** signal can be used to avoid zombies
- Windows having a fused fork-exec, which is very unlike Linux

Conclusion

- Processes are running programs
- OSes provide a rich set of syscalls to deal with processes
- Make sure you understand all the examples
  - Better if you experiment yourself by compiling/ playing with them
- Fork-exec in UNIX / CreateProcess in Windows
- Let's look at Sample Homework Assignment #2
- Onward to Inter-Process-Communication…