# Process

Copyright ©: University of Illinois CS 241 Staff

# What is a Process?

- Definition: an executable instance of a program
  - A process is the *context* (the information/data) maintained for an executing program
  - How is a program different from a process?
    - a program is a passive collection of instructions;
    - a process is the actual execution of those instructions; each process has a state to keep track of its execution

- Process provides each program with two key abstractions
  - Logical control flow
    - Each program seems to have exclusive use of the CPU
  - Private virtual address space
    - Each program seems to have exclusive use of main memory

Copyright ©: University of Illinois CS 241 Staff

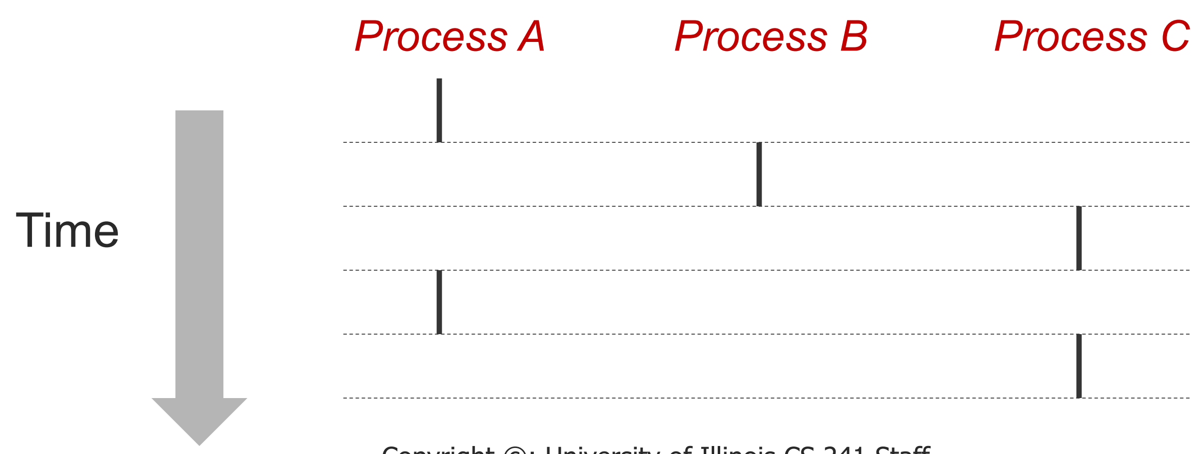
# What is a Process?

- How are these illusions maintained?
  - Process executions interleaved (multitasking) or run on separate cores
  - Address spaces managed by virtual memory system

- Unix processes
  - Process #1 is known as the 'init' process (root of the process hierarchy)
  - Each process has a unique identifier

# Concurrent Processes

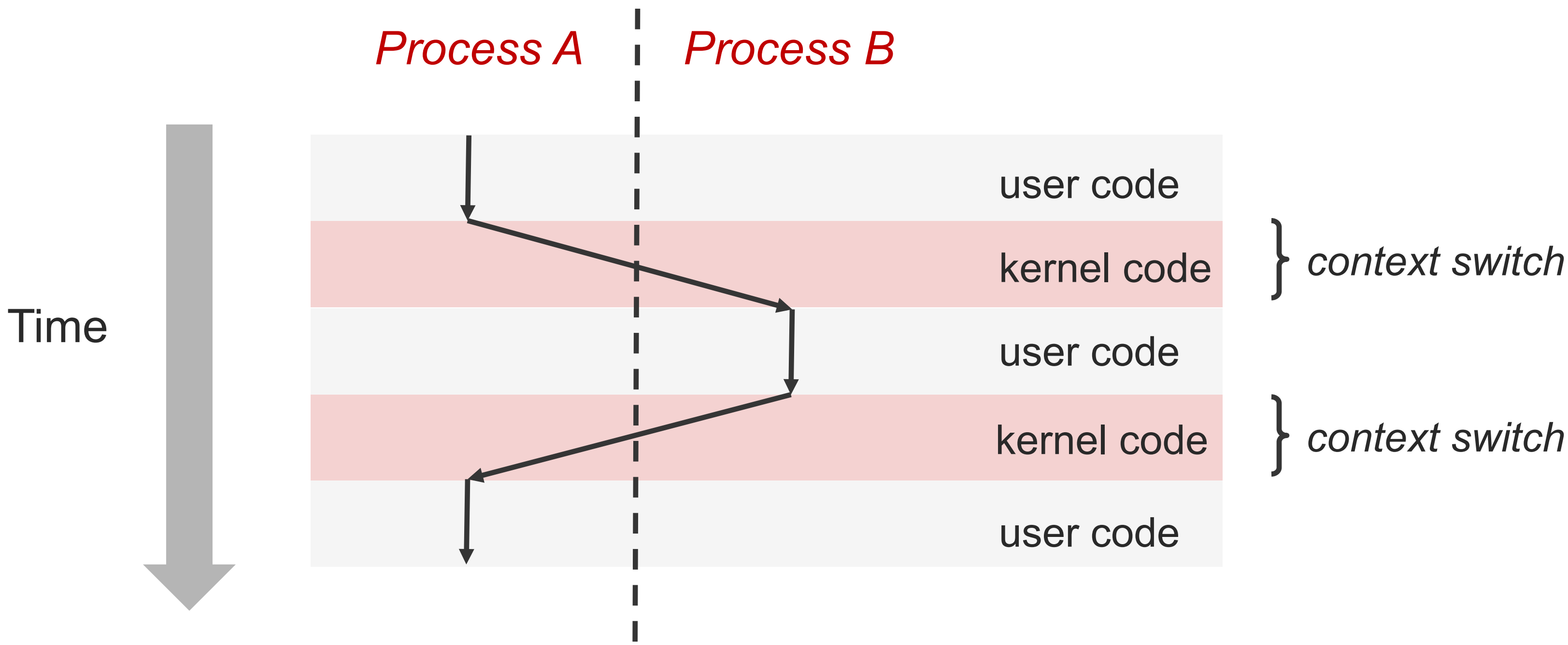
- Two processes run concurrently (are concurrent) if their flows overlap in time
  - Otherwise, they are sequential
- Examples (running on single core)
  - Concurrent: A & B, A & C
  - Sequential: B & C




Copyright ©: University of Illinois CS 241 Staff

# Context Switching

- Processes are managed by the kernel
- Control passes from one process to another via a context switch




Copyright ©: University of Illinois CS 241 Staff

# What makes up a process?

- Program code
- Machine registers
- Global data
- Stack
- Open files
- An environment

# Process Context

| | |
|---|---|
| Process ID (`pid`) | unique integer |
| Parent process ID (`ppid`) | unique integer |
| Current directory | |
| File descriptor table | |
| Environment | |
| Pointer to program code | |
| Pointer to data | Mem for global vars |
| Pointer to stack | Mem for local vars |
| Pointer to heap | Dynamically allocated memory |
| Execution priority | |
| Signal information | |

Copyright ©: University of Illinois CS 241 Staff

# Unix Processes

- Virtual address space
  - The virtual address space is the memory that contains the code to execute as well as the process stack and data
- Process Descriptor: data structure in the kernel to keep track of that process
  - Virtual address space map
  - Current status of the process
  - Execution priority of the process
  - Resource usage of the process
  - Current signal mask
  - Owner of the process

Copyright ©: University of Illinois CS 241 Staff

# Know your process

- Know your process id

```
pid_t myid = getpid()
```

- Know your parent

```
pid_t myparentid = getppid()
```

Copyright ©: University of Illinois CS 241 Staff

# Creating a Process – `fork()`

```
#include <sys/types.h>
#include <unistd.h>
pid_t fork(void);
```

- Create a child process
  - The child is an (almost) exact copy of the parent
  - The new process and the old process both continue in parallel from the statement that follows the `fork()`
- Returns:
  - To child
    - 0 on success
  - To parent
    - process ID of the child process
    - -1 on error, sets `errno`

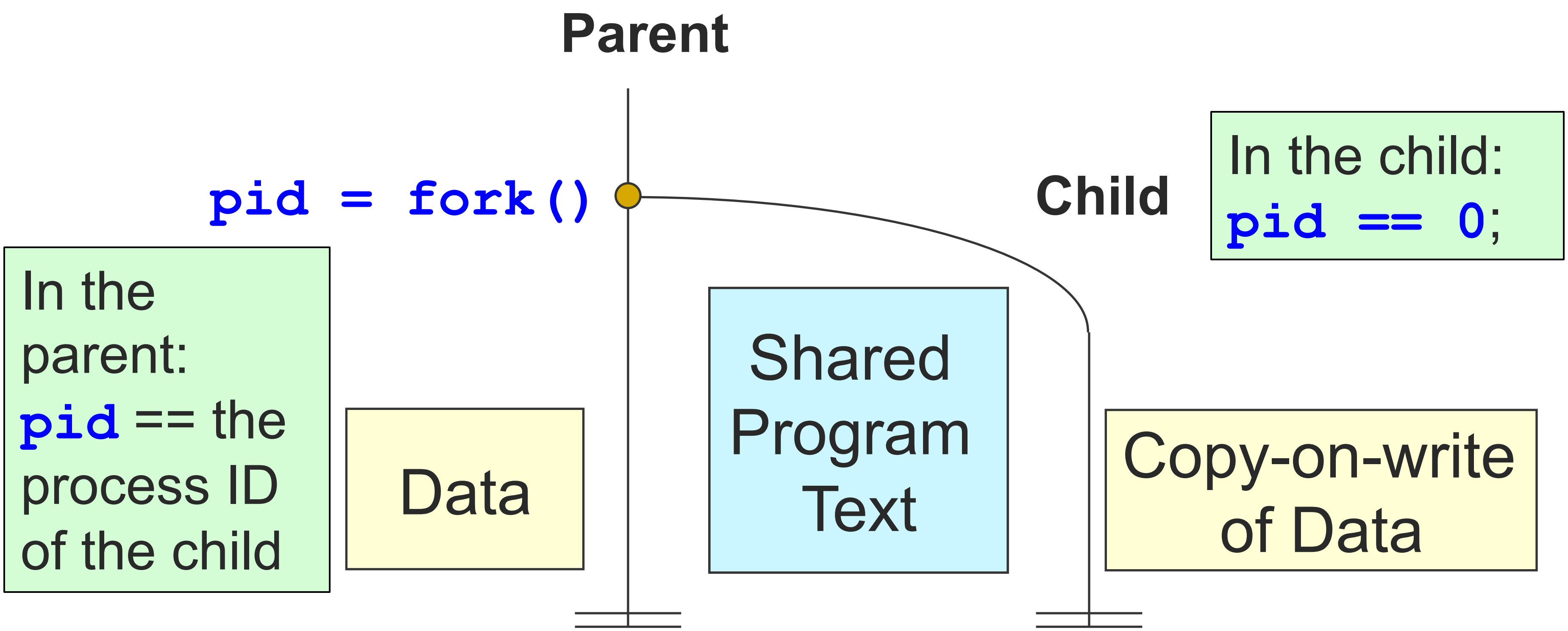
# Understanding `fork()`

- Fork is interesting (and often confusing) because it is called once but returns twice

```
pid_t pid = fork();
if (pid == 0) {
   printf("hello from child\n");
} else {
   printf("hello from parent\n");
}
```

Copyright ©: University of Illinois CS 241 Staff

# Creating a Process – `fork()`




A program can use this `pid` difference to do different things in the parent and child

# An Example

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

int main() {
    pid_t pid;
    int i;
    pid = fork();
    if( pid > 0 ) {        /* parent */
        for( i=0; i < 1000; i++ )
                printf("\t\t\t PARENT %d\n", i);
    } else { /* child */
        for(i=0; i < 1000; i++)
                printf( "CHILD %d\n", i );
    }
    return 0;
}
```

What will the output be?

Copyright ©: University of Illinois CS 241 Staff

# An Example

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

int main() {
    pid_t pid;
    int i;
    pid = fork();
    if( pid > 0 ) {        /* parent */
        for( i=0; i < 1000; i++ )
                printf("\t\t\t PARENT %d\n", i);
    } else { /* child */
        for(i=0; i < 1000; i++)
                printf( "CHILD %d\n", i );
    }
    return 0;
}
```

- Both processes start with same state
  - Each of them has a private virtual address space
  - Including an identical copy of open file descriptors
- Relative ordering of parent/child print statements (and so variable manipulations) is undefined

What will the output be?

Copyright ©: University of Illinois CS 241 Staff

# Possible Output

```
CHILD 0
CHILD 1
CHILD 2
                PARENT 0
                PARENT 1
                PARENT 2
                PARENT 3
CHILD 3
CHILD 4
                PARENT 4
          :
```

Copyright ©: University of Illinois CS 241 Staff

# Possible Output

- Switching between parent and child depends on many factors
  - Machine load, OS CPU scheduler
- I/O buffering affects amount of shown output
- Output interleaving is nondeterministic
  - Cannot determine output by looking at code

Copyright ©: University of Illinois CS 241 Staff

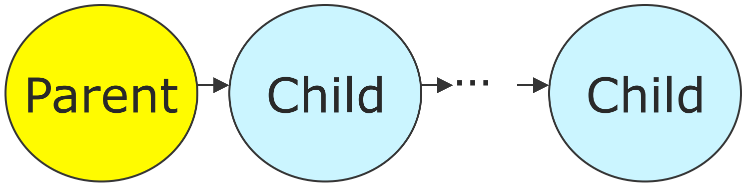
# Chain and Fan

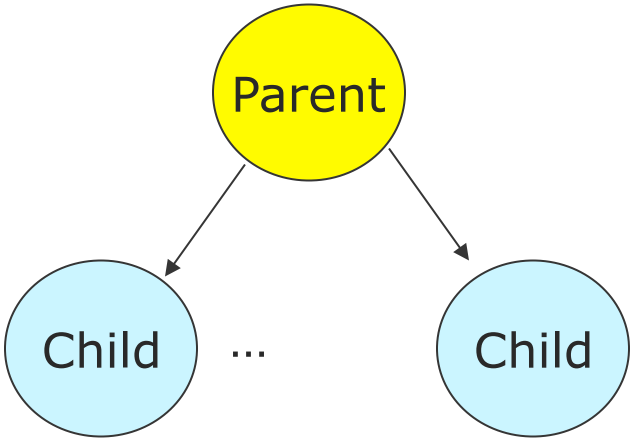
**Chain**

- Write code to make chain

**Fan**

- Code to make N children of one parent process



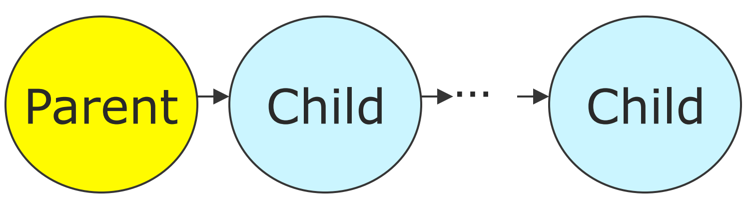




Copyright ©: University of Illinois CS 241 Staff

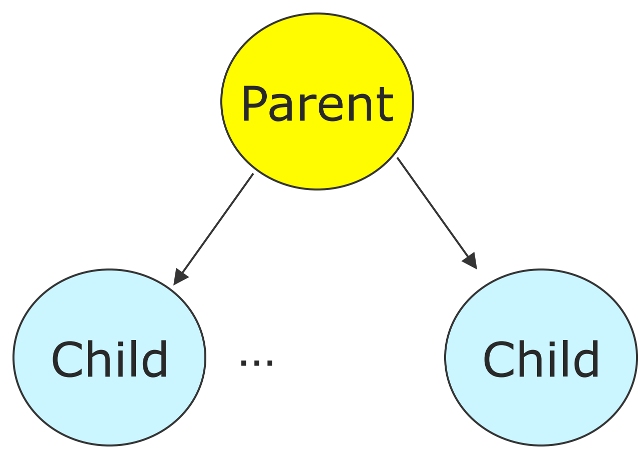
# Chain and Fan

**Chain**

```
pid_t childpid;
for (i=1;i<n;i++)
  if (childpid = fork())
      break;
```




**Fan**

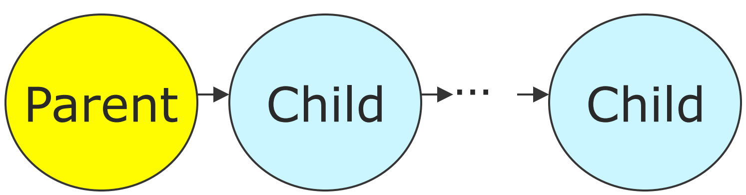



Copyright ©: University of Illinois CS 241 Staff

# Chain and Fan

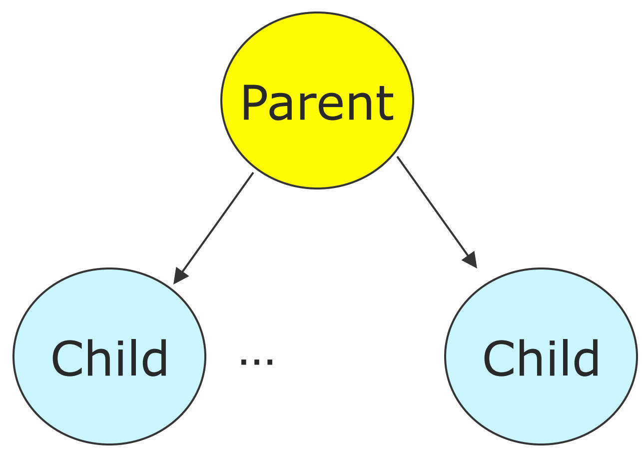
**Chain**

```
pid_t childpid;
for (i=1;i<n;i++)
  if (childpid = fork())
     break;
```




**Fan**

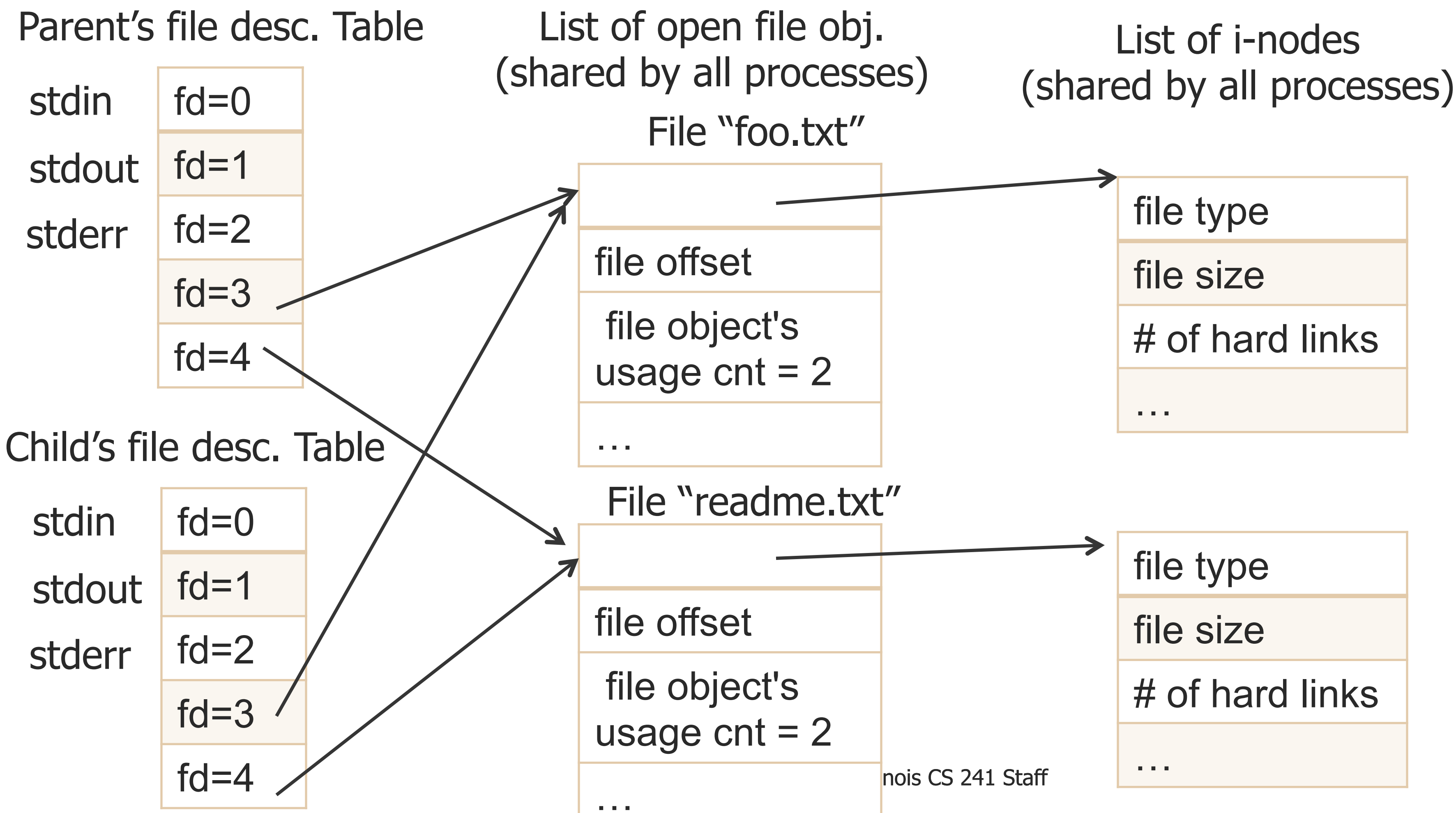
```
pid_t childpid;
for (i=1;i<n;i++)
  if ((childpid=fork()) <=0)
     break;
```




Copyright ©: University of Illinois CS 241 Staff

# Child process inherits parent's open files

- Parent forks **after** opening files foo.txt and readme.txt



# When a process terminates

- When a child process terminates:
  - Open files are flushed and closed
  - Child’s resources are de-allocated
    - File descriptors, memory, semaphores, file locks, …
  - Parent process is notified via signal SIGCHLD
  - Exit status is available to parent via wait()

# Process Termination

- Voluntary termination
  - Normal exit
    - return zero from **`main()`**
    - **`exit(0)`**
  - Error exit
    - **`exit(1)`**
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - **`kill`** procID, end task

Copyright ©: University of Illinois CS 241 Staff

# `exit()` Example

`void exit(int status)`

- Exits a process
- Normally return with status 0

`atexit()`

- Registers functions to be executed upon exit

```
void cleanup(void) {
        printf("cleaning up\n");
}

int main() {
        atexit(cleanup);
        fork();
        exit(0);
}
```

# Zombies

- What happens on termination?
  - When process terminates, still consumes system resources
  - Entries in various tables & info maintained by OS
- Called a “zombie”
  - Living corpse, half alive and half dead

# Zombies

- Reaping
  - Performed by parent on terminated child (using **`wait`** or **`waitpid`**)
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then child will be reaped by **`init`** process (**`pid == 1`**)
  - So, only need explicit reaping in long-running processes
    - e.g., shells and servers

Copyright ©: University of Illinois CS 241 Staff

# Zombie Example

```
void forktest() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1); /* Infinite loop */
    }
}
```

# Zombie Example

```
void forktest() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating C
               getpid());
        exit(0);
    } else {
        printf("Running Paren
               getpid());
        while (1); /* Infinit
    }
}
```

```
Linux> ./forktest 7 &
[1] 8992
Terminating Child, PID = 8993
Running Parent, PID = 8992
Linux> ps
  PID TTY          TIME CMD
 8992 pts/1    00:00:06 forktest
 8993 pts/1    00:00:00 forktest <defunct>
 8994 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
Linux> kill 8992
[1]+  Terminated              ./forktest
Linux> ps
  PID TTY          TIME CMD
 9004 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
```

- `ps` shows child process as “defunct”
- Killing parent allows child to be reaped by `init`

Copyright ©: University of Illinois CS 241 Staff