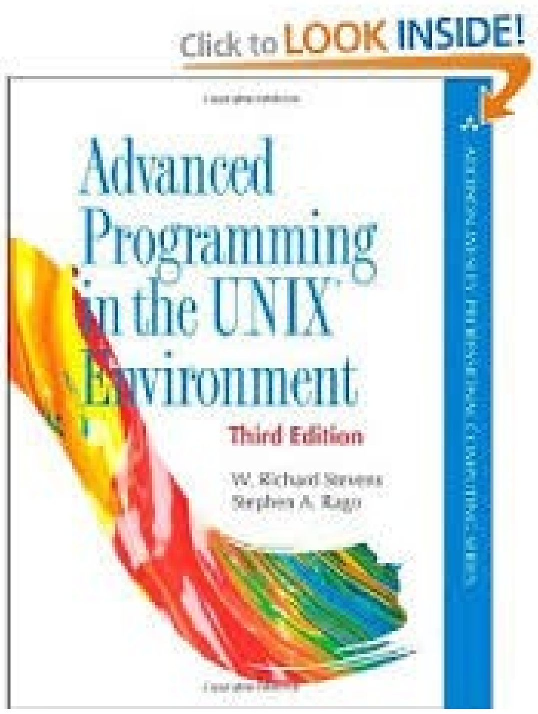
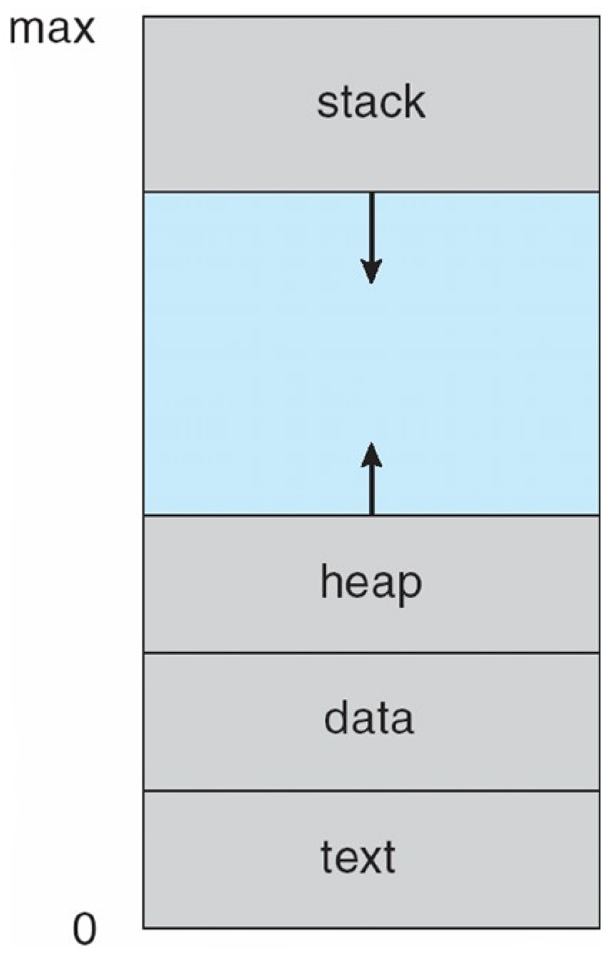
Where are we?

- **-C fundamentals**
  -Memory
- **-Unix Fundamentals**
- **-Files**
- **-Processes (today)**
- **-Threads**
- **-Interprocess communication**
- **-Parallel Computing**
- **-Networking**
- **-Optional Advanced Topics**
-



Chapter 8. Process Control

Process in Memory



- Text
  - Program code
- Data
  - Global variables
- Stack
  - Temporary data
  - Function parameters, return addresses, local variables
- Heap
  - Dynamically allocated memory

Memory Layout of a C Program

Text segment: the machine instructions that the CPU executes. Usually, the text segment is sharable so that only a single copy needs to be in memory for frequently executed programs, such as text editors, the C compiler, the shells, and so on. Also, the text segment is often read-only, to prevent a program from accidentally modifying its instructions.

Initialized data segment: usually called simply the data segment, containing variables that are specifically initialized in the program. For example, the C declaration

Int maxcount = 99;

appearing outside any function causes this variable to be stored in the initialized data segment with its initial value.

•

Memory Layout of a C Program

Uninitialized data segment, often called the "bss" segment, named after an ancient assembler operator

that stood for "block started by symbol." Data in this segment is initialized by the kernel to arithmetic 0 or null pointers before the program starts executing. The C declaration

```
long   sum[1000];
```

appearing outside any function causes this variable to be stored in the uninitialized data segment.

Stack, where automatic variables are stored, along with information that is saved each time a function is called. Each time a function is called, the address of where to return to and certain information about the caller's environment, such as some of the machine registers, are saved on the stack. The newly called function then allocates room on the stack for its automatic and temporary variables. This is how recursive functions in C can work. Each time a recursive function calls itself, a new stack frame is used, so one set of variables doesn't interfere with the variables from another instance of the function.

Heap, where dynamic memory allocation usually takes place. Historically, the heap has been located between the uninitialized data and the stack.

Memory Layout of a C Program

The allocation routines are usually implemented with the sbrk(2) system call. This system call expands (or contracts) the heap of the process.

A sample implementation of malloc and free is given in Section 8.7 of Kernighan and Ritchie

Although sbrk can expand or contract the memory of a process, most versions of malloc and free never decrease their memory size. The space that we free is available for a later allocation, but the freed space is notusually returned to the kernel; that space is kept in the malloc pool.

Process Concept

- An OS executes a variety of programs:
  - Batch system — jobs
  - Time-shared systems — user programs or tasks

- Textbooks uses the terms *job* and *process* almost interchangeably

- Process — a program in execution
  - Process execution must progress in sequential fashion

Process Concept

- Definition: A process is an instance of a running program
- One of the most profound ideas in computer science
- Not the same as “program” or “processor”
- 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

Process Concept

- Process A process is the context (the information/data) maintained for an executing program
- An executable instance of a program
- A program can have many processes
- Each process has a unique identifier
- Unix processes
  - Process #1 is known as the 'init' process (root of the process hierarchy)

    %ps aux

Process Lifetime

- Some processes run from system boot to shutdown
- Servers & Daemons
  - (e.g. Apache httpd server)
- Most processes come and go rapidly, as
- tasks start and complete
  - 'unit of work' on a modern computer
- A process can die a premature, even horrible death (say, due to a crash)

Process ID

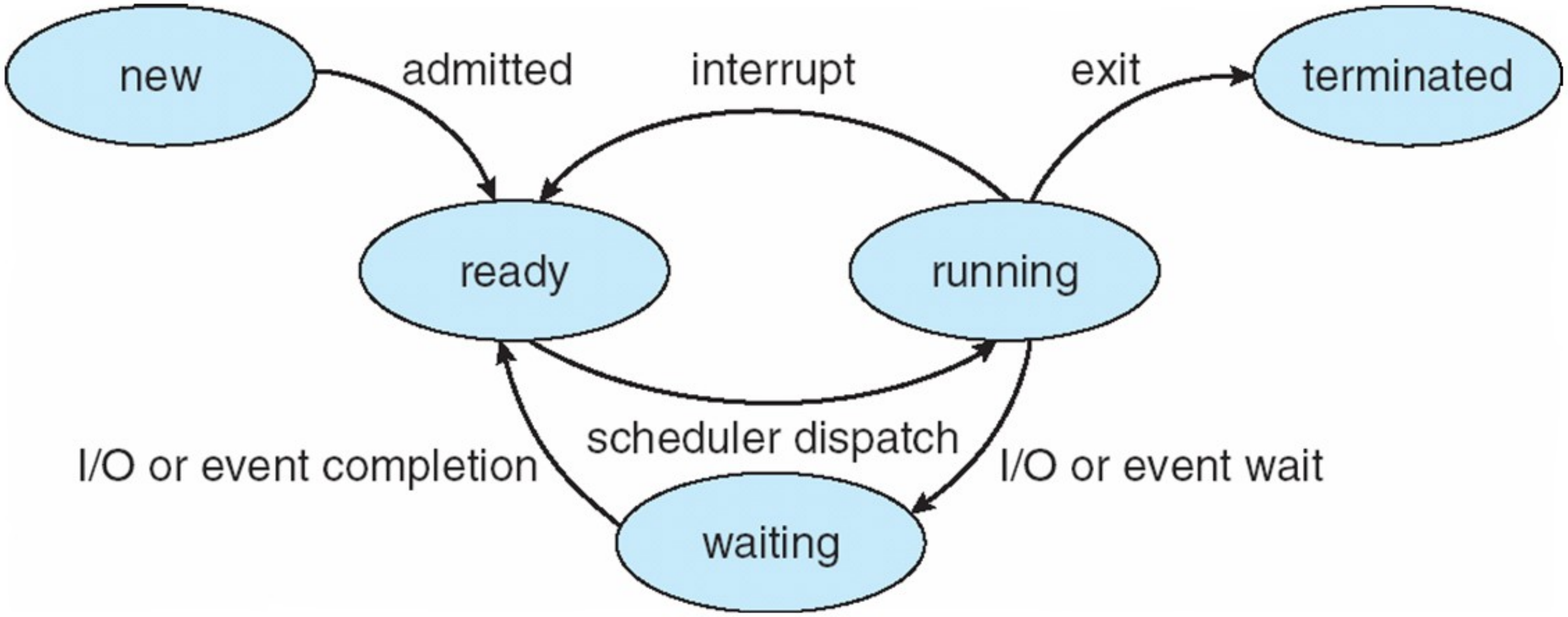
```
#include <unistd.h>

pid_t getpid(void);
pid_t getppid(void);
```

- Process ID’s are guaranteed to be unique and identify a particular executing process with a non-negative integer.
- Certain processes have fixed, special identifiers. They are:
  - swapper, process ID 0 – responsible for scheduling
  - init, process ID 1 – bootstraps a Unix system, owns orphaned processes
- pagedaemon, process ID 2 – responsible for the VM system (some Unix systems)
- %ps aux
- example: pid.c

Process State

- As a process executes, it changes state
  - **New**: The process is being created
  - **Running**: Instructions are being executed
  - **Waiting**: The process is waiting for some event to occur
  - **Ready**: The process is waiting to be assigned to a processor
  - **Terminated**: The process has finished execution

Diagram of Process State



Process Creation

- On creation, process needs resources
  - CPU, memory, files, I/O devices
- Get resources from the OS or from the parent process
  - Child process is restricted to a subset of parent resources
  - Prevents many processes from overloading system

Creating a Process – fork()

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

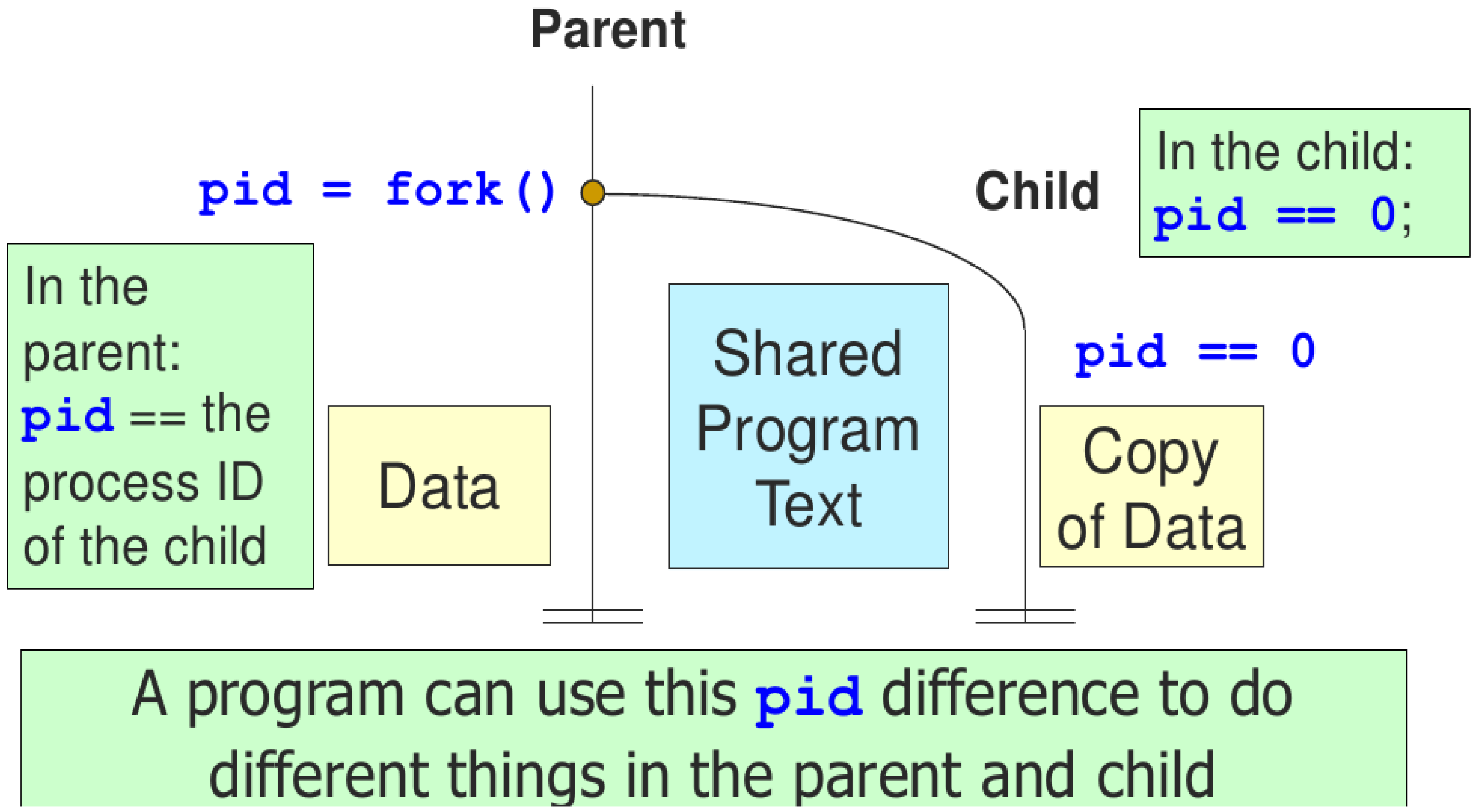
- Create a child process
- The child is an (almost) exact copy of the parent
- The new process and the old process both continue in parallel from the statement that follows the fork()
- Returns:
  - To child
    - 0 on success
- To parent
  - process ID of the child process
  - -1 on error, sets errno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

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");
}
```

Understanding fork()



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

Understanding fork()

- Parent

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

pid = m

```
hello from parent
```

- Child

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

pid = 0

```
hello from child
```

Fork example 1

```
void fork1() {
    int x = 1;
    pid_t pid = fork();
    if (pid == 0) {
        printf("Child has x = %d\n", ++x);
    } else {
        printf("Parent has x = %d\n", --x);
    }
    printf("Bye from process %d with x = %d\n",
           getpid(), x);
}
```

- Both processes start with same state
  - Each has private copy
  - Including shared output file descriptor
- Relative ordering of their print statements (and so variable manipulations) is undefined

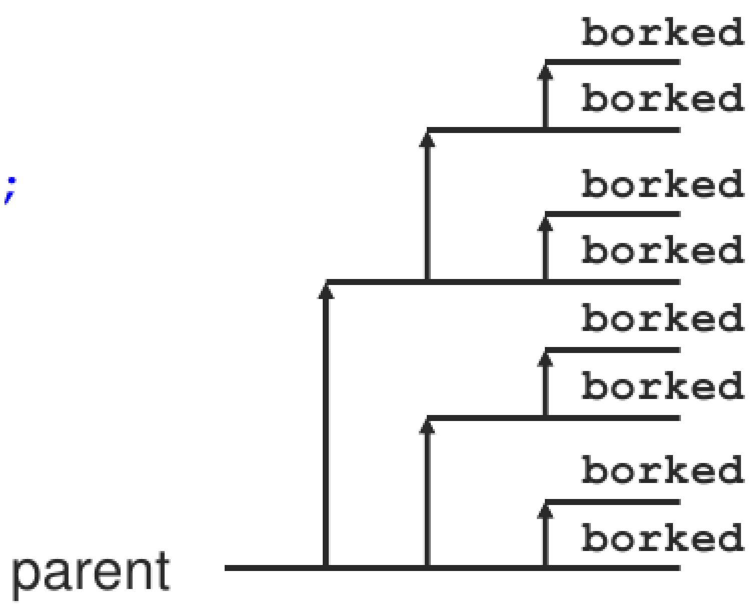
Fork example 1

```
#define bork fork

void forkbork()
{
     bork(); bork(); bork();
     printf(”borked\n");
}
```

Fork example 2

```
#define bork fork

void forkbork()
{
    bork(); bork(); bork();
    printf("borked\n");
}
```

Three consecutive forks



Fork example 3

```
void fork3()
{
    printf("L0\n");
    fork();
    printf("L1\n");
    fork();
    printf("L2\n");
    fork();
    printf("Bye\n");
}
```

Fork example 5

```
void fork5()
{
    printf("L0\n");
    if (fork() == 0) {
       printf("L1\n");
       if (fork() == 0) {
           printf("L2\n");
           fork();
       }
    }
    printf("Bye\n");
}
```

Process Termination

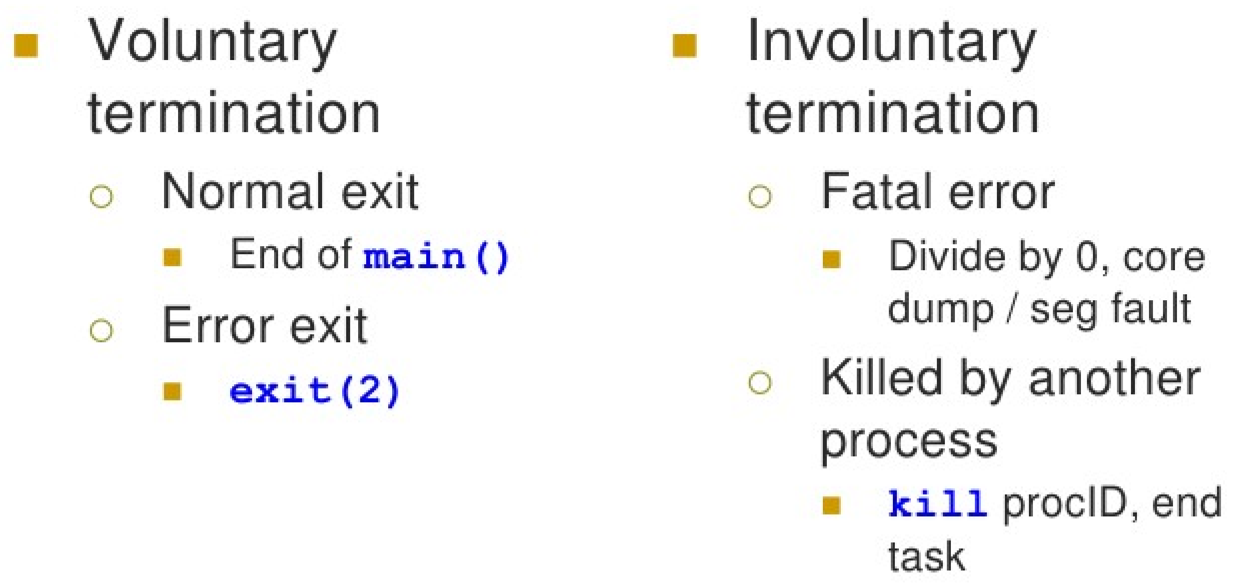
- Upon completion of last statement
  - A process automatically asks the OS to delete it
  - All of the child’s resources are de-allocated

    - Child process may return output to parent process
- Other termination possibilities:
  - Aborted by parent process

    - Child has exceeded its usage of some resources

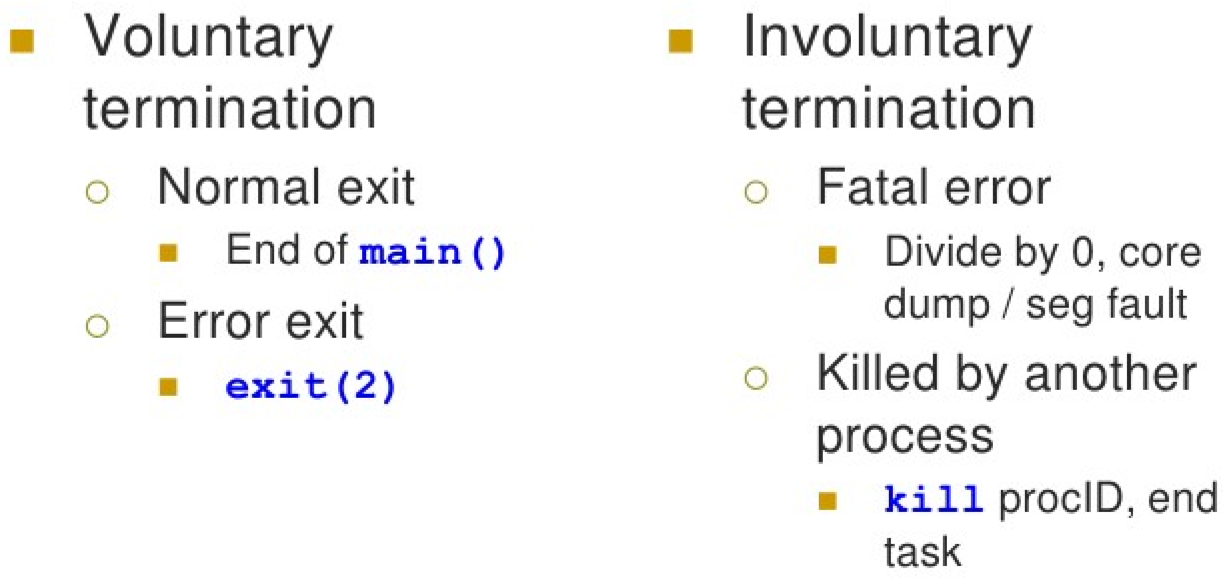
    - Task assigned to child is no longer required
  - Parent is exiting and OS does not allow child to continue without parent
-

Process Termination



- Voluntary termination
  - Normal exit
    - End of `main()`
  - Error exit
    - `exit(2)`
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - `kill` procID, end task

exit() Example

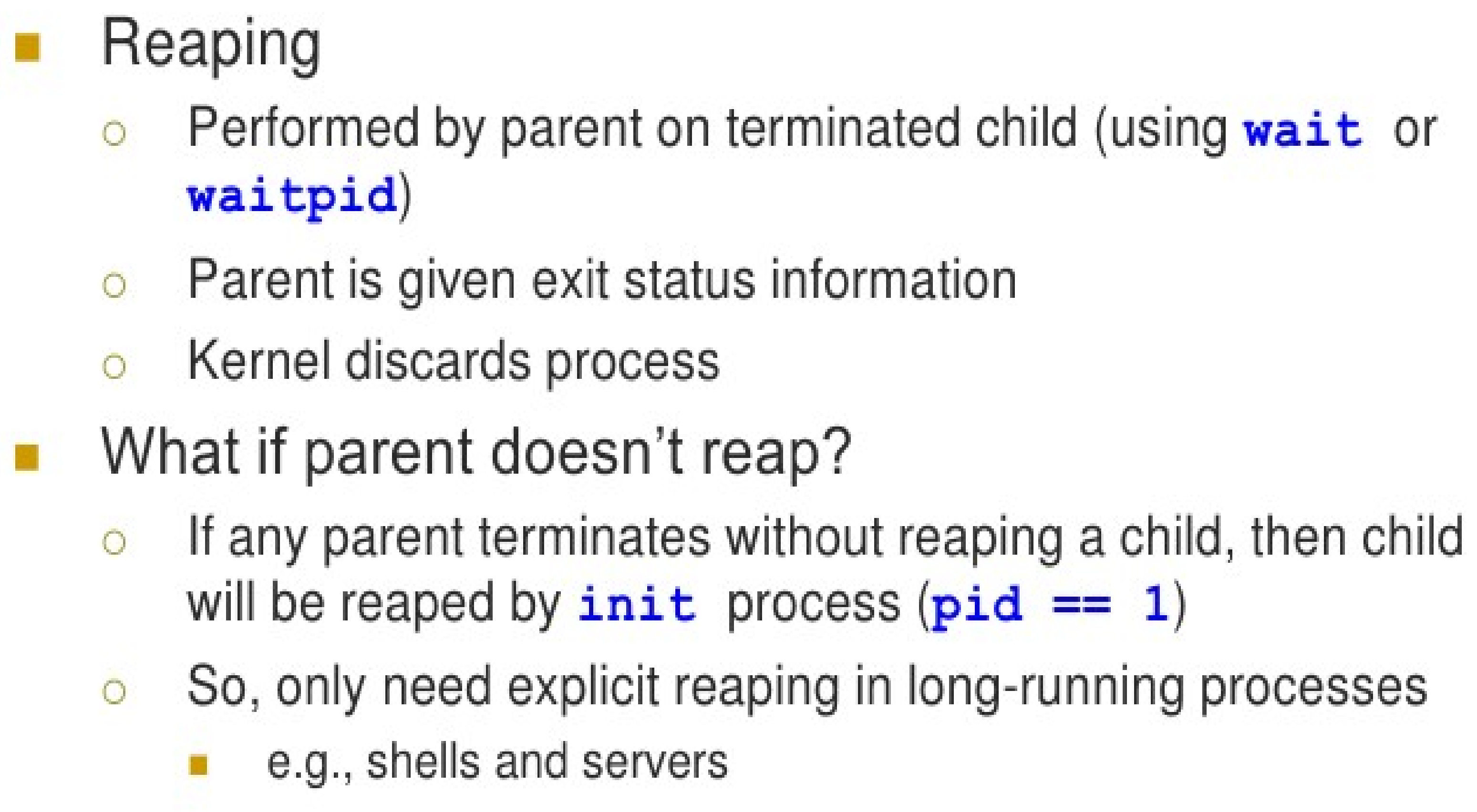


- Voluntary termination
  - Normal exit
    - End of `main()`
  - Error exit
    - `exit(2)`
- Involuntary termination
  - Fatal error
    - Divide by 0, core dump / seg fault
  - Killed by another process
    - `kill` procID, end task

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

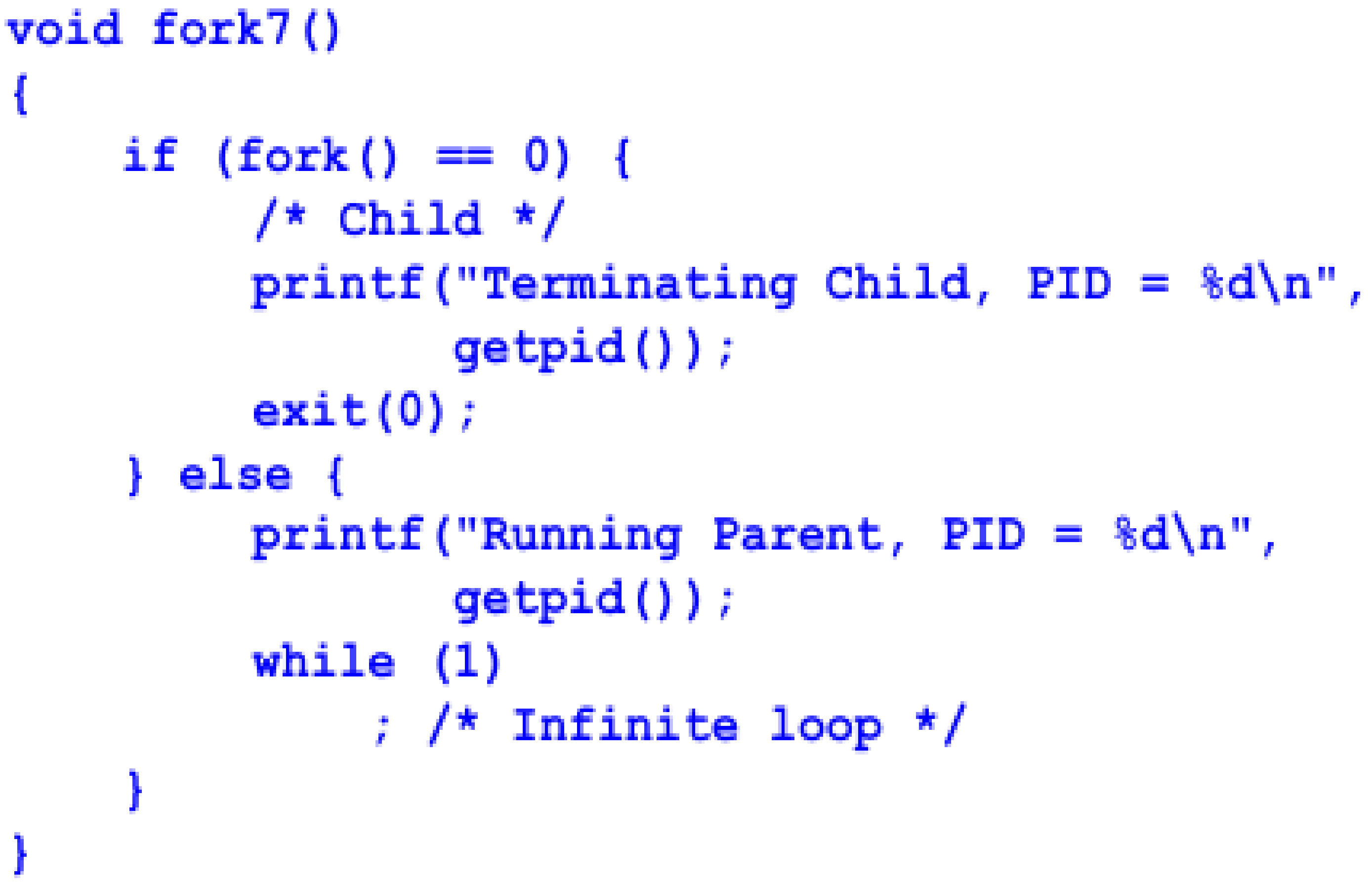


- Reaping
  - Performed by parent on terminated child (using `wait` or `waitpid`)
  - Parent is given exit status information
  - Kernel discards process
- What if parent doesn't reap?
  - If any parent terminates without reaping a child, then child will be reaped by `init` process (`pid == 1`)
  - So,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

Zombie Example

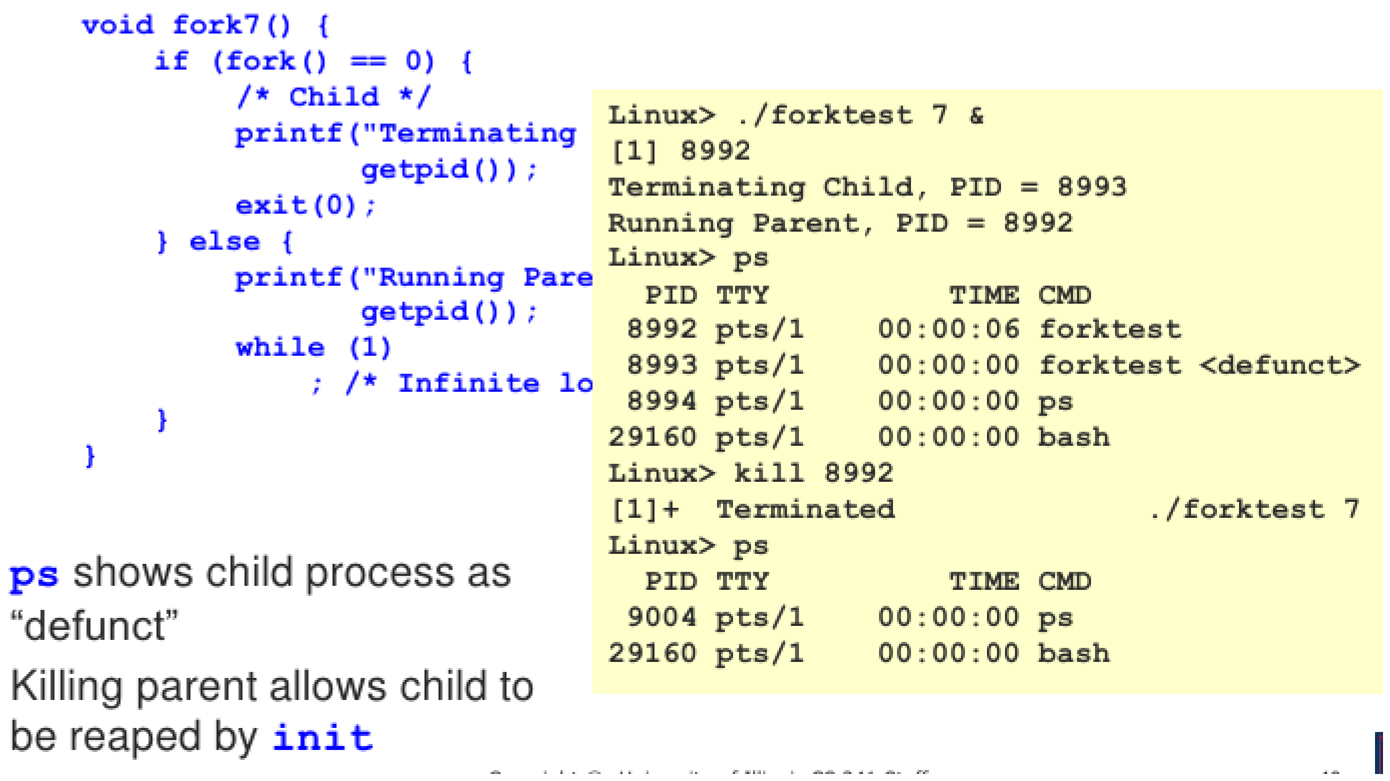
```
void fork7()
{
    if (fork() == 0) {
        /* Child */
        printf("Terminating Child, PID = %d\n",
               getpid());
        exit(0);
    } else {
        printf("Running Parent, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    }
}
```

Zombie Example



```
void fork7()
{
    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 fork7() {
    if (fork() == 0) {
        /* Child */
        printf("Terminating
               getpid());
        exit(0);
    } else {
        printf("Running Pare
               getpid());
        while (1)
            ; /* Infinite lo
    }
}
```

```
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 7
Linux> ps
  PID TTY          TIME CMD
 9004 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
```

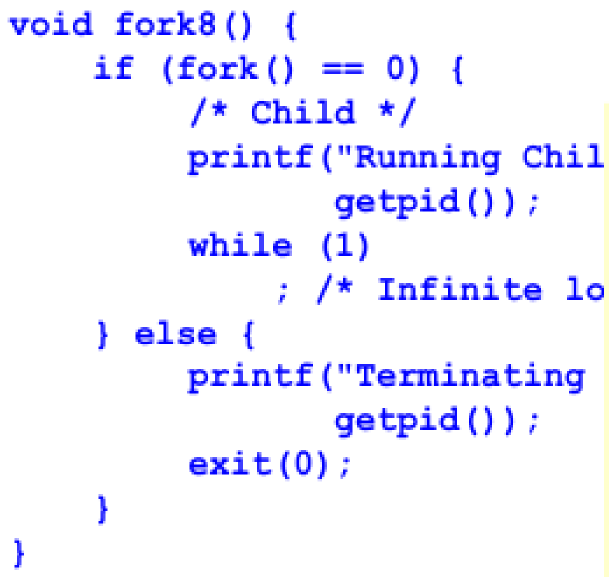
`ps` shows child process as "defunct"

Killing parent allows child to be reaped by `init`

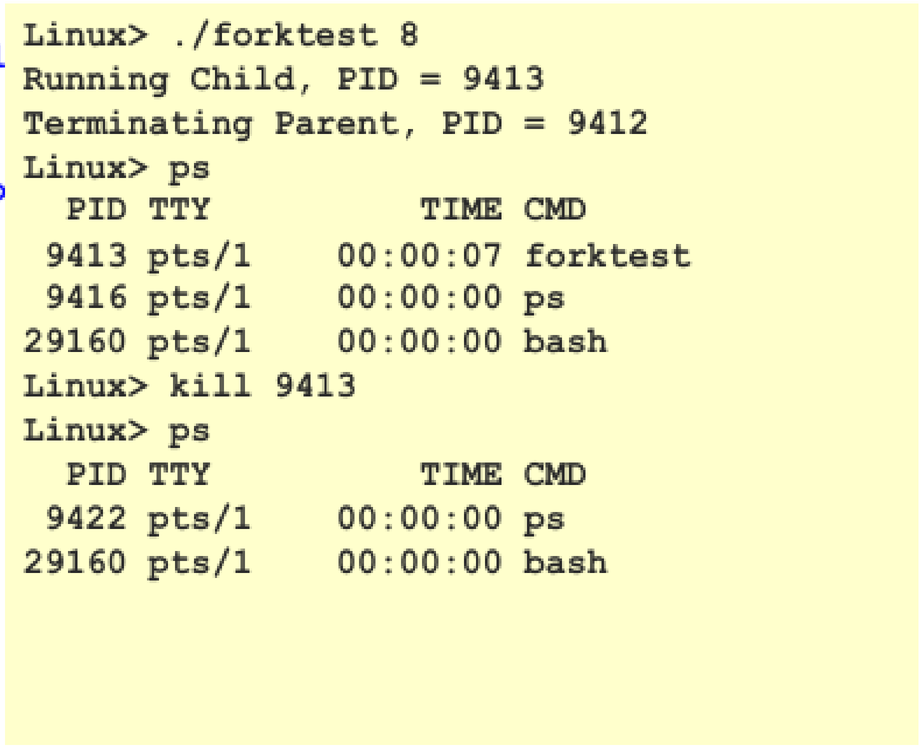
Orphan Example

```
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Child, PID = %d\n",
               getpid());
        while (1)
            ; /* Infinite loop */
    } else {
        printf("Terminating Parent, PID = %d\n",
               getpid());
        exit(0);
    }
}
```

Orphan Example



```
void fork8() {
    if (fork() == 0) {
        /* Child */
        printf("Running Chil
               getpid());
        while (1)
            ; /* Infinite lo
    } else {
        printf("Terminating
               getpid());
        exit(0);
    }
}
```



```
Linux> ./forktest 8
Running Child, PID = 9413
Terminating Parent, PID = 9412
Linux> ps
  PID TTY          TIME CMD
 9413 pts/1    00:00:07 forktest
 9416 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
Linux> kill 9413
Linux> ps
  PID TTY          TIME CMD
 9422 pts/1    00:00:00 ps
29160 pts/1    00:00:00 bash
```

Child process still active even though parent has terminated

Must kill explicitly, or else will keep running indefinitely

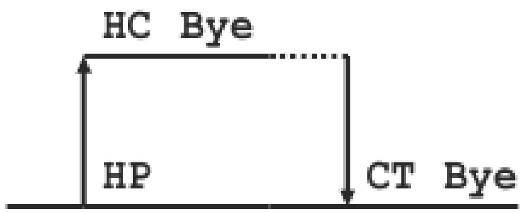
Waiting for a child to finish

```
#include <sys/types.h>
#include <wait.h>
pid_t wait(int *status);
```

- Suspend calling process until child has finished
- Allow parent to reap child
- Returns:
  - Process ID of terminated child on success
  - -1 on error, sets `errno`
- Parameters:
  - `status`: status information set by `wait` and evaluated using specific macros defined for `wait`.

Waiting for a child to finish

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
    }
    else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
    exit();
}
```



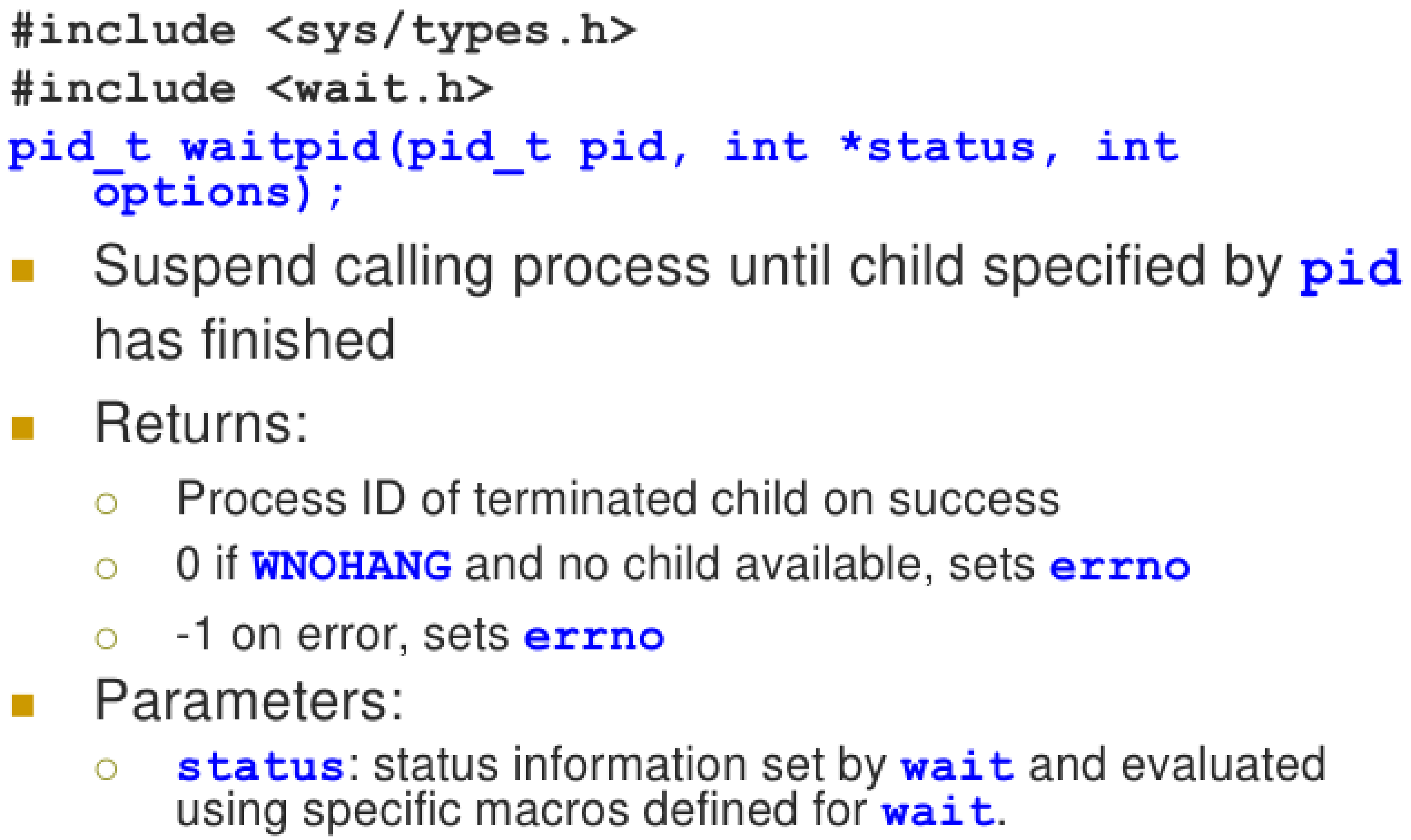
Waiting for any child to finish

```
void fork10() {
    pid_t pid[N];
    int i;
    int child_status;
    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = 0; i < N; i++) {
        pid_t wpid = wait(&child_status);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
           wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminate abnormally\n", wpid);
    }
```

If multiple children complete, they are taken in an arbitrary order

Can use macros **WIFEXITED** and **WEXITSTATUS** to get information about exit status

Waiting for a child to finish waitpid()



```
#include <sys/types.h>
#include <wait.h>
pid_t waitpid(pid_t pid, int *status, int
    options);
```

- Suspend calling process until child specified by `pid` has finished
- Returns:
  - Process ID of terminated child on success
  - 0 if `WNOHANG` and no child available, sets `errno`
  - -1 on error, sets `errno`
- Parameters:
  - `status`: status information set by `wait` and evaluated using specific macros defined for `wait`.

Waiting for a child to finish waitpid()

- Suspend calling process until child specified by `pid` has finished
- Parameters:
  - `pid`:
    - < -1: wait for any child process whose process group ID is equal to the absolute value of `pid`.
    - -1 wait for any child process (same as `wait`)
    - 0 wait for any child process whose process group ID is equal to that of the calling process.
    - > 0 wait for the child whose process ID is equal to the value of `pid`.

Waiting for a child to finish waitpid()

- Suspend calling process until child specified by **`pid`** has finished
- Parameters:
  - **`pid`**:
    - < -1: wait for any child process whose process group ID is equal to the absolute value of **`pid`**.
    - -1 wait for any child process (same as **`wait`**)
    - 0 wait for any child process whose process group ID is equal to that of the calling process.
    - > 0 wait for the child whose process ID is equal to the value of **`pid`**.

Waiting for a child to finish waitpid()

```
void fork11() {
    pid_t pid[N];
    int i;
    int child_status;

    for (i = 0; i < N; i++)
    if ((pid[i] = fork()) == 0)
        exit(100+i); /* Child */
    for (i = N-1; i >= 0; i--) {
        pid_t wpid = waitpid(pid[i], &child_status, 0);
    if (WIFEXITED(child_status))
        printf("Child %d terminated with exit status %d\n",
           wpid, WEXITSTATUS(child_status));
    else
        printf("Child %d terminated abnormally\n", wpid);
    }
}
```

exec(3) functions

```
#include <unistd.h>

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

int execv(const char *pathname, char * const argvp[]);

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

int execve(const char *pathname, char * const argvp[], char * const envp[]);

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

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

The `exec()` family of functions are used to completely replace a running process with a a new executable.

- if it has a v in its name, argv's are a vector: `const * char argv[]`
- if it has an l in its name, argv's are a list: `const char *arg0, ... /* (char *) 0 */`
- if it has an e in its name, it takes a `char * const envp[]` array of environment variables
- if it has a p in its name, it uses the `PATH` environment variable to search for the file

C Program Forking Separate Process

```
int main()
{
      pid_t  pid;
      /* fork another process */
      pid = fork();
      if (pid < 0) { /* error occurred */
            fprintf(stderr, "Fork Failed");
            exit(-1);
      }
      else if (pid == 0) { /* child process */
            execlp("/bin/ls", "ls", NULL);
      }
      else { /* parent process */
            /* parent will wait for the child to
complete */
            wait (NULL);
            printf ("Child Complete");
            exit(0);
      }
}
```

C Program Forking Separate Process

Simple ls
Simple shell

References

The previous and following slides are mostly taken from:

- http://courses.engr.illinois.edu/cs241/sp2012/
  http://www.cs.stevens.edu/~jschauma/810D/
-