Reminder from last week:

Simplest version of DayOfYear

```
class DayOfYear {
public:
    void output();
    int month;
    int day;
};
void DayOfYear::output() {
    cout << "month = " << month
         << ",  day = " << day << endl;
 }
```

- Like a struct with an added method
  - All parts public
  - Clients access month, day directly

What's wrong with DayOfYear?

- Most important: data are exposed to users
- Why is that a problem?
- Two major reasons:
  - No way to insure consistent object states – e.g. user could `birthday.month = 74;` // huh?
  - Developer can't change data names/meanings – e.g. can't change int to string for month, can't save Date instead of month, day, …
- What's the solution (in C++)?

An access specifier: `private`

- Private members of a class can only be referenced within the definitions of member functions (and `friends` – later)
  - If the program tries to access a private member, the compiler gives an error message
- Private members can be data or functions
  - Should have public set methods to change data
  - Need public get methods to access the data
- Btw: default for class is private (public for struct)

Better class DayOfYear

```
class DayOfYear {
public:
    void input( );
    void output( );
    void set(int new_month, int new_day);
    int get_month( );
    int get_day( );
private:
    void check_date( );
    int month;
    int day;
};
```

DISPLAY 10.4

Creating and assigning (=) objects

- Declaring an object *creates the object*

```
DayOfYear today, tomorrow;
   // two objects are created on stack
```

- Different if declaring pointers (or references)

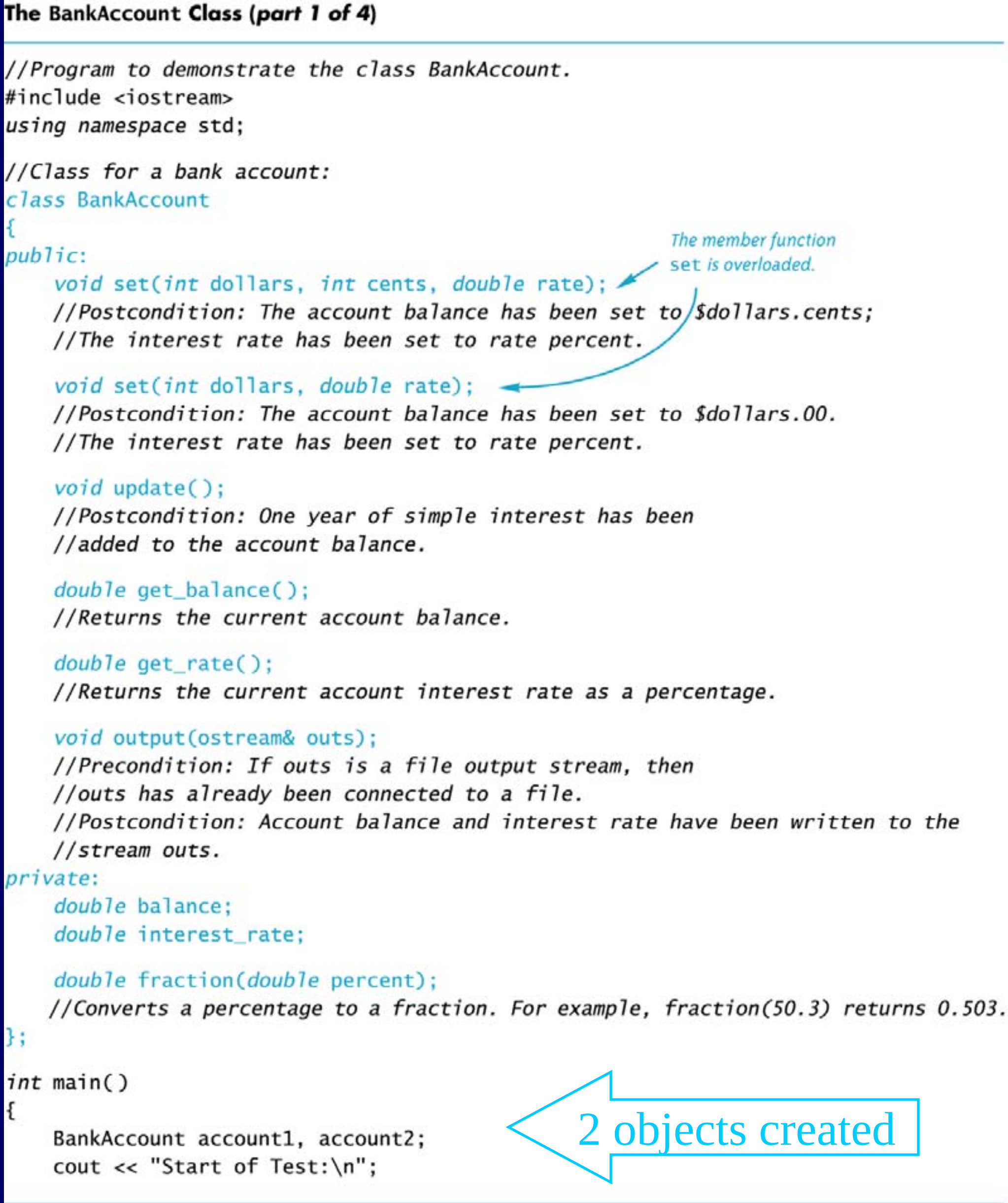
```
DayOfYear *soon, &r = today; // no object
soon = new DayOfYear; // now object on heap
```

- Assignment operator *copies object’s data*

```
r = *soon; // no new object–just copy on stack
   // original (today) object data overwritten
```

Another class example: BankAccount

- Has operations appropriate for a bank account (implemented with public member functions)
  - And a private utility function
- Stores an account balance and an interest rate



The BankAccount Class (*part 1 of 4*)

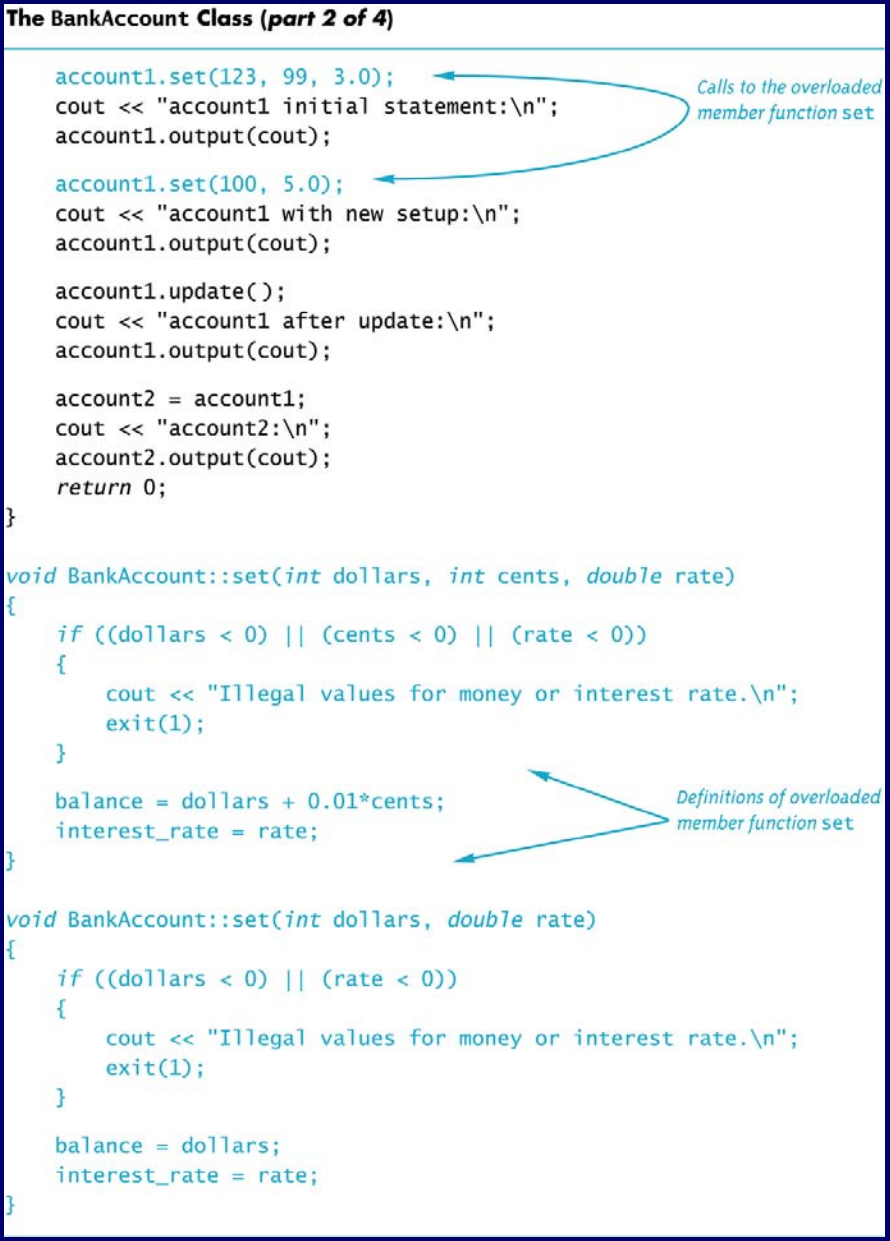
```
//Program to demonstrate the class BankAccount.
#include <iostream>
using namespace std;

//Class for a bank account:
class BankAccount
{
public:
    void set(int dollars, int cents, double rate);
    //Postcondition: The account balance has been set to $dollars.cents;
    //The interest rate has been set to rate percent.

    void set(int dollars, double rate);
    //Postcondition: The account balance has been set to $dollars.00.
    //The interest rate has been set to rate percent.

    void update();
    //Postcondition: One year of simple interest has been
    //added to the account balance.

    double get_balance();
    //Returns the current account balance.

    double get_rate();
    //Returns the current account interest rate as a percentage.

    void output(ostream& outs);
    //Precondition: If outs is a file output stream, then
    //outs has already been connected to a file.
    //Postcondition: Account balance and interest rate have been written to the
    //stream outs.
private:
    double balance;
    double interest_rate;

    double fraction(double percent);
    //Converts a percentage to a fraction. For example, fraction(50.3) returns 0.503.
};

int main()
{
    BankAccount account1, account2;
    cout << "Start of Test:\n";
```

The member function set is overloaded.

2 objects created

Method overloading – `BankAccount::set`

- A method's signature includes its name and its parameter list
- Can overload a name like set with a different parameter list
  - Number, types, order



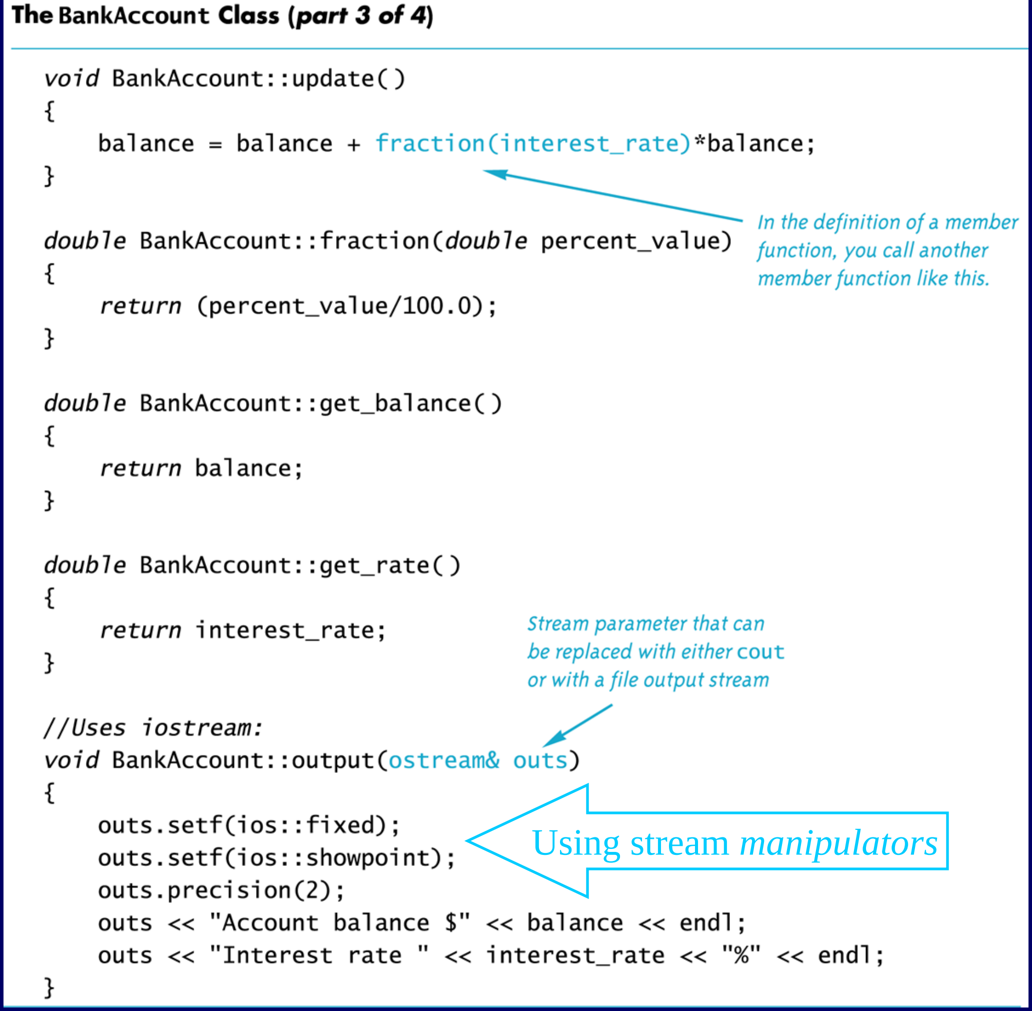
The BankAccount Class (*part 2 of 4*)

```
    account1.set(123, 99, 3.0);
    cout << "account1 initial statement:\n";
    account1.output(cout);

    account1.set(100, 5.0);
    cout << "account1 with new setup:\n";
    account1.output(cout);

    account1.update();
    cout << "account1 after update:\n";
    account1.output(cout);

    account2 = account1;
    cout << "account2:\n";
    account2.output(cout);
    return 0;
}

void BankAccount::set(int dollars, int cents, double rate)
{
    if ((dollars < 0) || (cents < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars + 0.01*cents;
    interest_rate = rate;
}

void BankAccount::set(int dollars, double rate)
{
    if ((dollars < 0) || (rate < 0))
    {
        cout << "Illegal values for money or interest rate.\n";
        exit(1);
    }

    balance = dollars;
    interest_rate = rate;
}
```

Calls to the overloaded member function set

Definitions of overloaded member function set

More implementing BankAccount



The BankAccount **Class (*part 3 of 4*)**

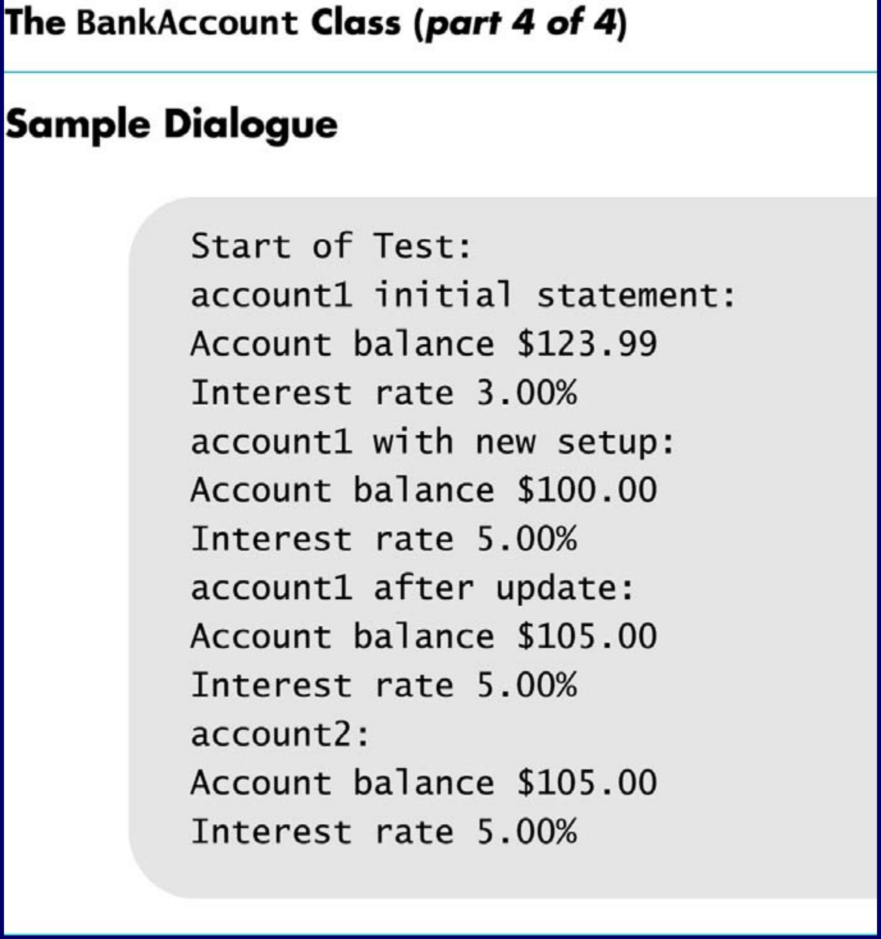
```
void BankAccount::update()
{
    balance = balance + fraction(interest_rate)*balance;
}

double BankAccount::fraction(double percent_value)
{
    return (percent_value/100.0);
}

double BankAccount::get_balance()
{
    return balance;
}

double BankAccount::get_rate()
{
    return interest_rate;
}

//Uses iostream:
void BankAccount::output(ostream& outs)
{
    outs.setf(ios::fixed);
    outs.setf(ios::showpoint);
    outs.precision(2);
    outs << "Account balance $" << balance << endl;
    outs << "Interest rate " << interest_rate << "%" << endl;
}
```

In the definition of a member function, you call another member function like this.

Stream parameter that can be replaced with either cout *or with a file output stream*

Using stream *manipulators*

Sample BankAccount results



The BankAccount Class (*part 4 of 4*)

Sample Dialogue

```
Start of Test:
account1 initial statement:
Account balance $123.99
Interest rate 3.00%
account1 with new setup:
Account balance $100.00
Interest rate 5.00%
account1 after update:
Account balance $105.00
Interest rate 5.00%
account2:
Account balance $105.00
Interest rate 5.00%
```

// excerpts from main:

```
account1.set(123,99,3);
```

// called with all 3 arguments

```
account1.set(100,5);
```

// called other version of set

```
account1.update();
```

```
account2 = account1;
```

Q: What if account2.update()?

Constructors

- A constructor (a.k.a. ctor) is a member function
  - Usually declared public
- One is *always* called when an object is created
- Main purpose – initialize instance variables
  - Also useful to allocate resources if needed
- Constructor's name must be the name of the class
- A constructor cannot return a value
  - No return type, not even void

A BankAccount constructor

- Declare in public part of class definition

```
BankAccount(int dollars, int cents, double rate);
```

- Implement essentially like other methods

```
BankAccount::BankAccount(int dollars, int cents,
                         double rate) {

    if ((dollars < 0) || (cents < 0) || ( rate < 0 )) {
        cout << "Illegal values for money or rate\n";
        exit(1);
    }

    balance = dollars + 0.01 * cents;
    interest_rate = rate;
}
```

Constructor call is automatic

- May not invoke (i.e., call) it directly:

  `account1.BankAccount(10, 50, 2);` // ERROR
- Instead invoke indirectly
  - On stack: `BankAccount account1(10, 50, 2);`
  - Or free store: … `new BankAccount(10, 50, 2);`
- But class must have a matching constructor
  - e.g., `BankAccount()` if just `new BankAccount;`
    - Default constructor is called – but oops: ERROR if explicit constructor is defined and not overloaded!

Overloading and the default ctor

- Another possible BankAccount ctor:
  ```
  BankAccount (double balance, double interest_rate);
  ```
- Or can have either one of the following. Why not both?
  ```
  BankAccount (double balance);
  BankAccount (double interest_rate);
  ```
- Also either explicitly define default ctor:
  ```
  BankAccount ( );
  ```

  Or implicitly via default arguments in other ctors:
  ```
  BankAccount (double balance = 0.0);
  ```
- Tip: good idea to always include a default ctor even if there is no need to initialize variables
  - So clients can: `BankAccount checking, savings;`
  - Important for inheritance reasons too (a future topic)

Base/member initialization list

- An initialization section in a constructor definition provides an alternative way to initialize member variables

```
BankAccount::BankAccount() : balance(0), interest_rate(0)
{ } // still need a body (even if intentionally empty like this case)
```

  - Can use parameter names too – even if same name as member!
- Note: order of initialization matches the order in which the variables are declared in the class, not their order in the list
- *Must* use such a list for constants and reference variables (since references are always constant)
  - Also must use to initialize private data in a base class (later topic)
- *Should* always use for user-defined types if default ctor not appropriate – to avoid extra ctor (and destructor) calls

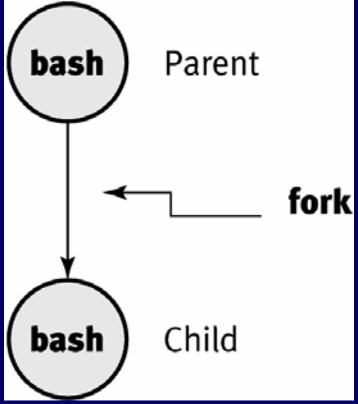
Starting Reading #3

Back to the OS – processes

Later: on to advanced class design (Savitch Chapter 11)

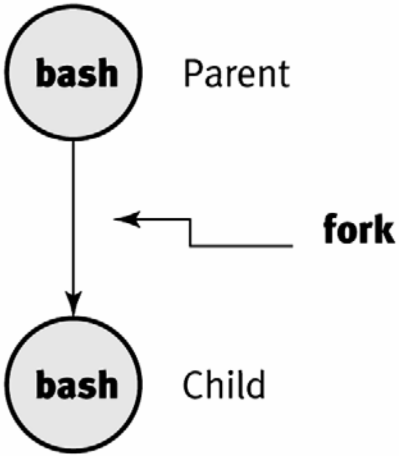
Processes

- A process is an executable, machine language program that the OS (Linux) has been asked to run
  - Copied to memory, and assigned a process ID (PID)
  - Scheduled for execution by the CPU
- Processes create other processes via system calls
  - A program (e.g., in C or C++) creates a new process and terminates itself with a call to `exec`
  - A program creates a child process by calling `fork`
  - e.g.: `$> ./myscript`
    - First line is: `#!/bin/bash`
    - bash runs (interprets script)

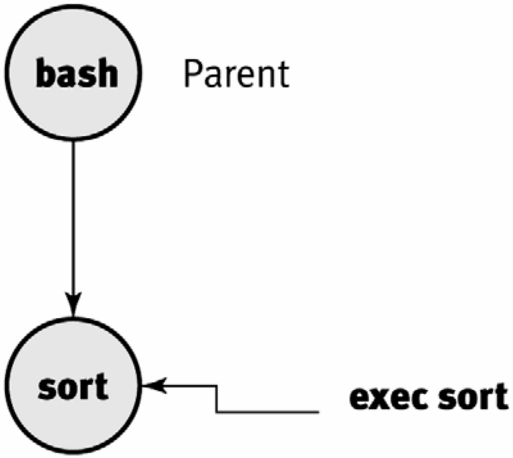


Steps to execute a program (sort)

Step 1: Shell uses **fork** to create a child



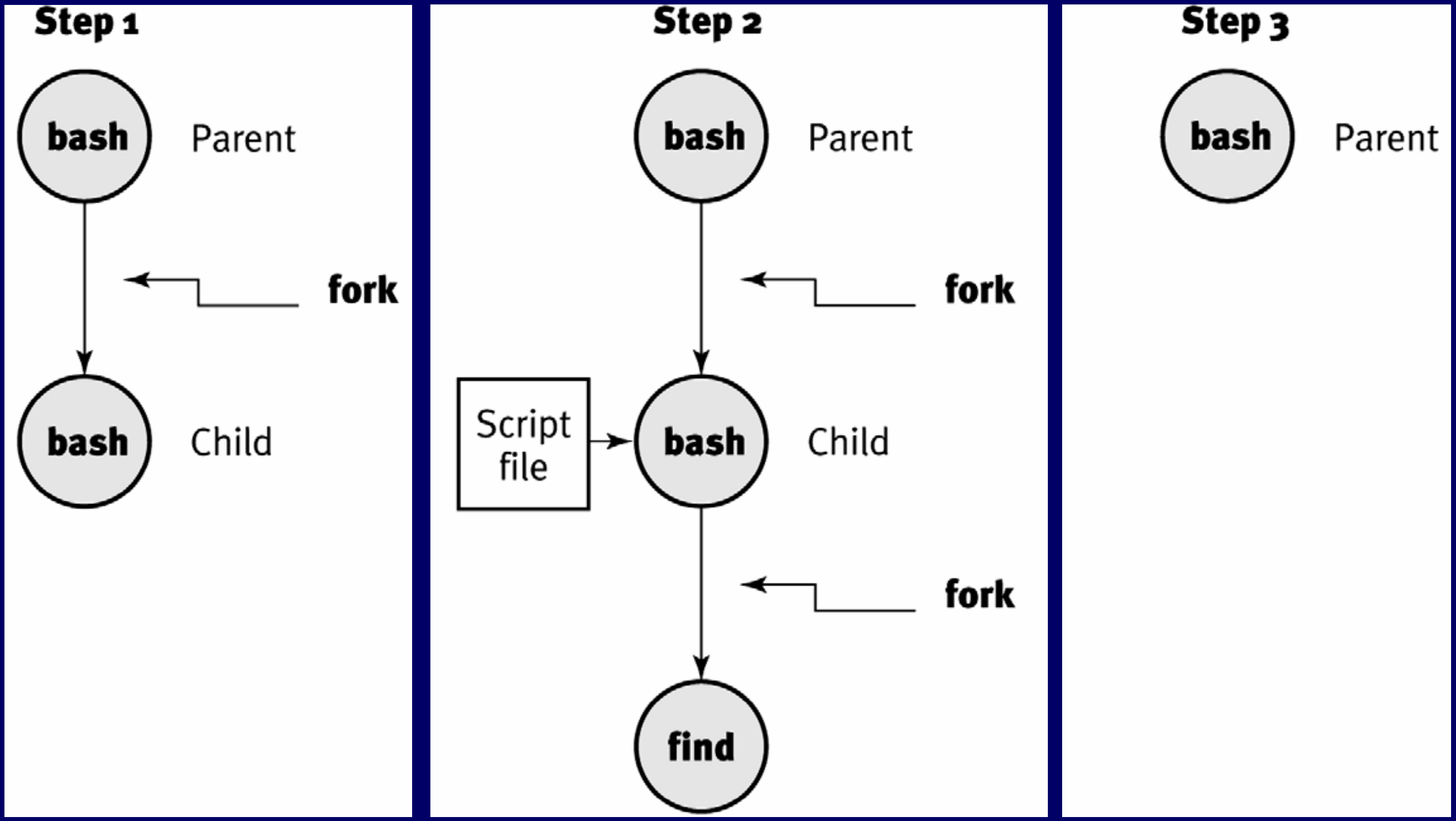
Step 2: Child uses **exec** to overwrite itself with the executable file corresponding to the **sort** command.



Step 3: **sort** starts execution while 'bash' waits for the command to finish. When **sort** finishes, the child process terminates and 'bash' starts execution again, waiting for the user to give it another command to execute.

bash
Parent

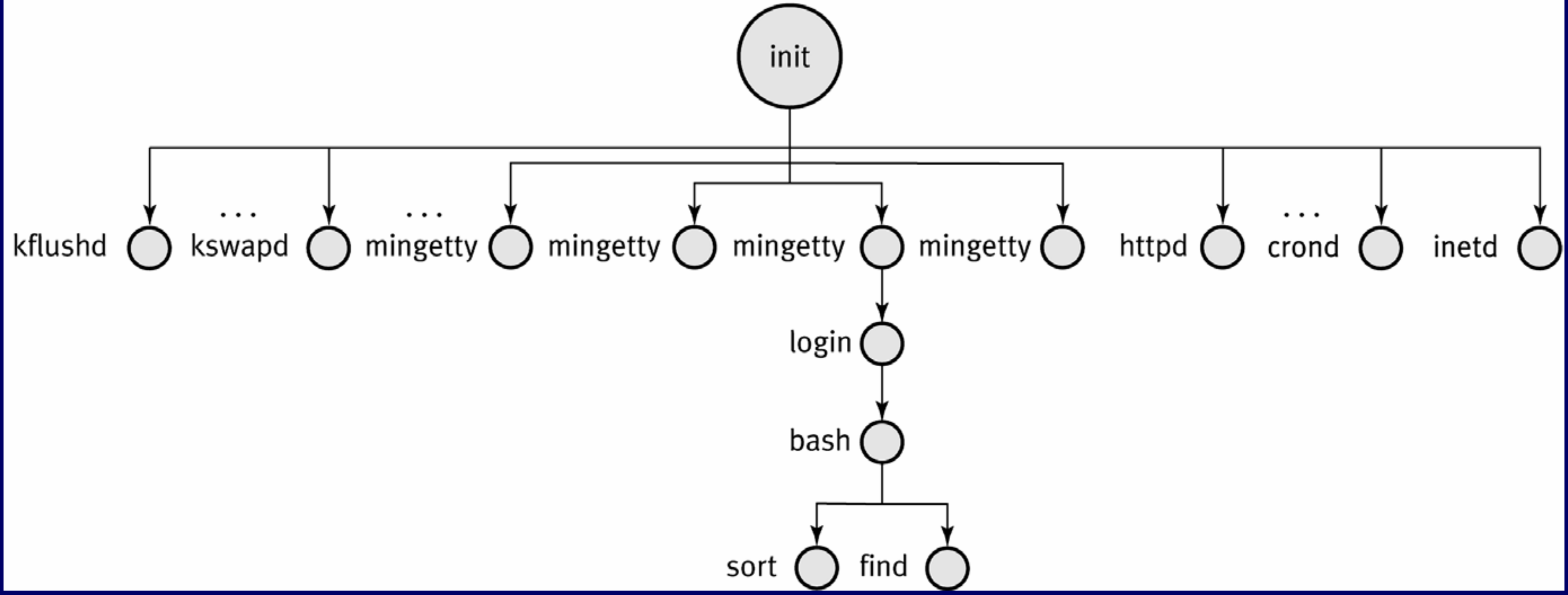
Steps to execute a shell script



Process hierarchy

- init – is PID 1, but all other processes have parents (so PPID)
  - The process hierarchy's depth is limited only by available virtual memory
- A process may control the execution of any of its descendants
  - Can suspend or resume it
  - Can alter its relative priority
  - Can even terminate it completely
- By default, terminating a process will terminate all of its descendants too
  - So terminating the root process will terminate the session

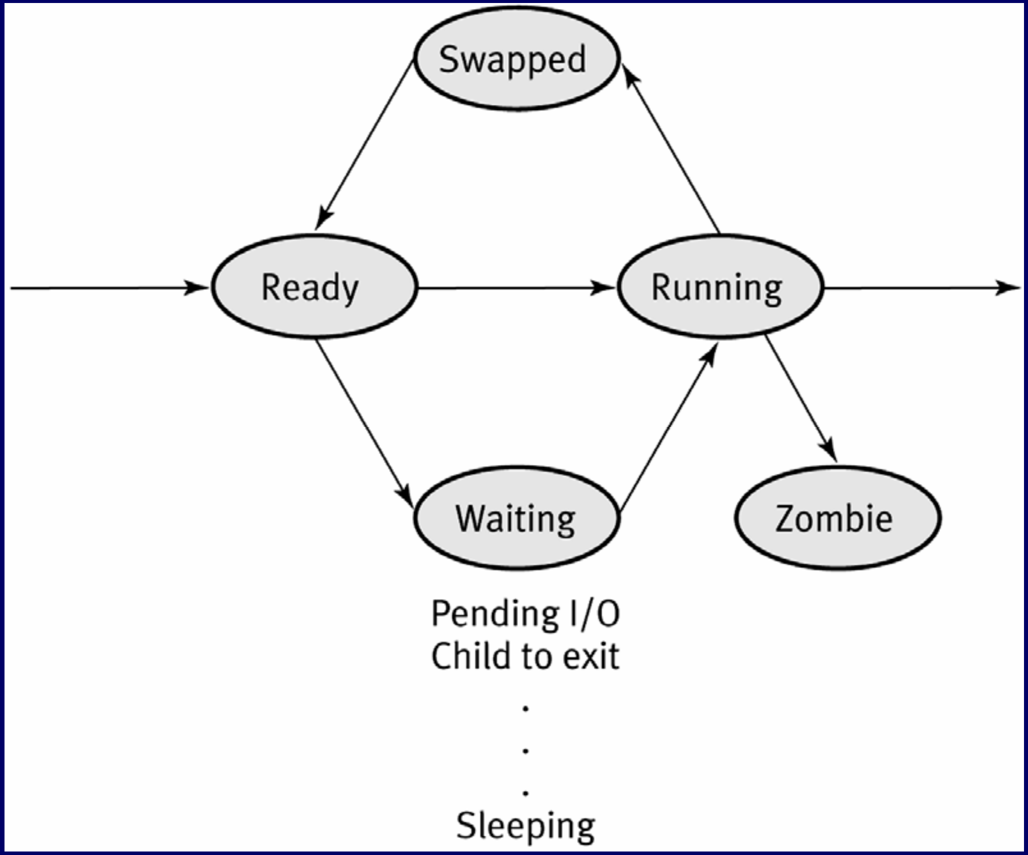
Example Linux process hierarchy



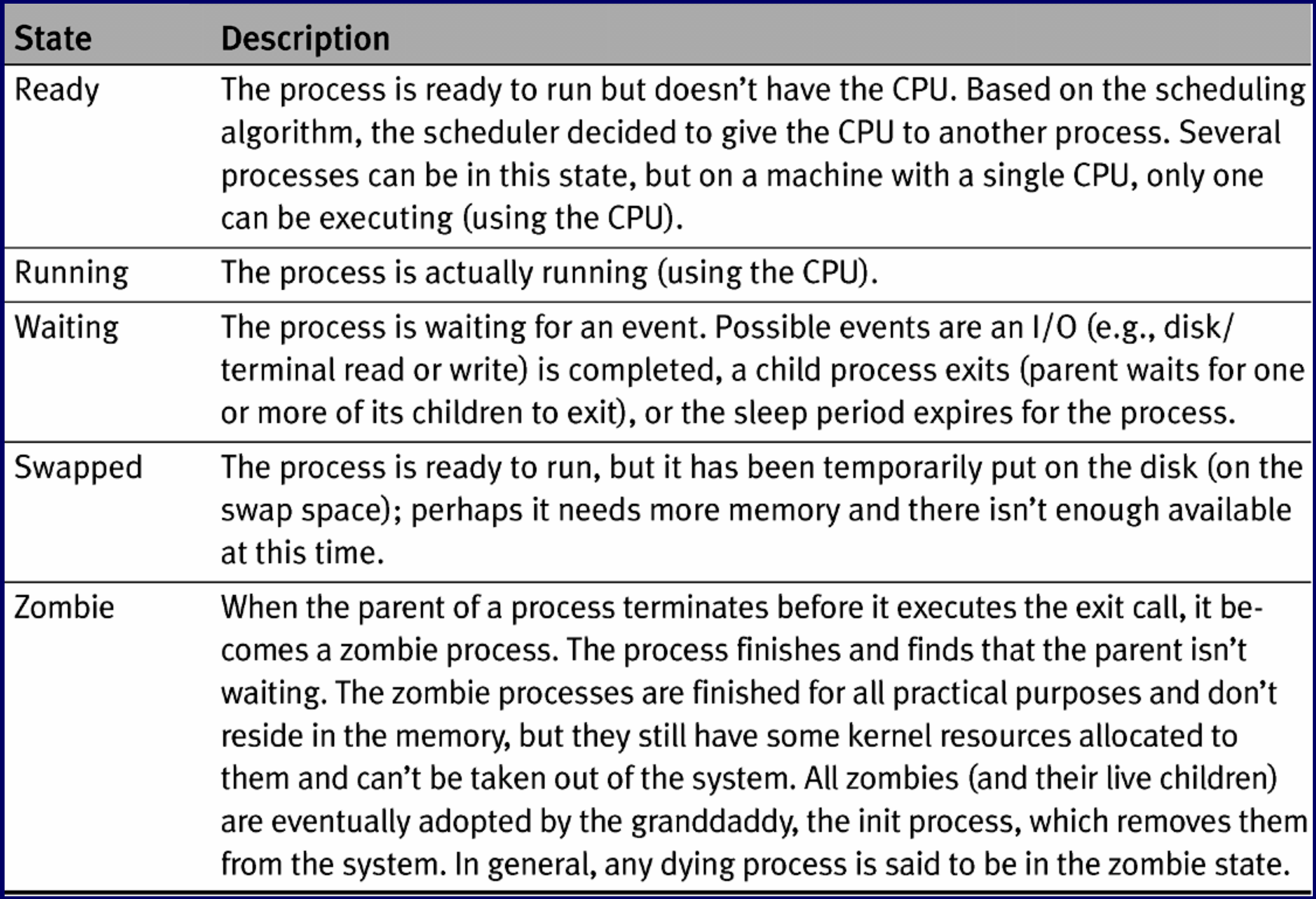
(From Linux-specific version of Sarwar et al. text)

Linux process states

- Just one process can be "running" at any one time
- OS has other processes in various states
- A process may be cycled through many states before it terminates

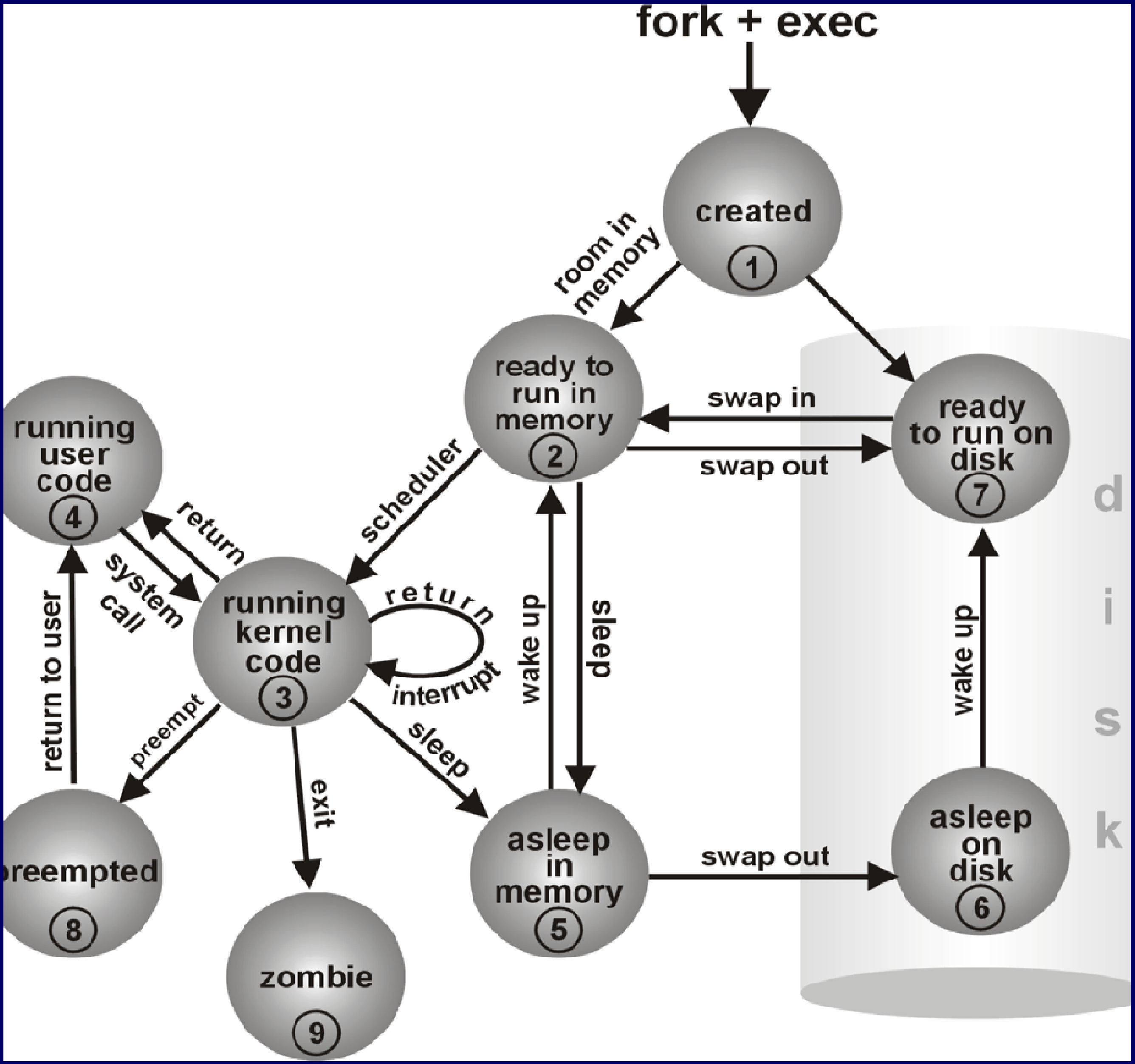


Meanings of Linux process states



State	Description
Ready	The process is ready to run but doesn't have the CPU. Based on the scheduling algorithm, the scheduler decided to give the CPU to another process. Several processes can be in this state, but on a machine with a single CPU, only one can be executing (using the CPU).
Running	The process is actually running (using the CPU).
Waiting	The process is waiting for an event. Possible events are an I/O (e.g., disk/ terminal read or write) is completed, a child process exits (parent waits for one or more of its children to exit), or the sleep period expires for the process.
Swapped	The process is ready to run, but it has been temporarily put on the disk (on the swap space); perhaps it needs more memory and there isn't enough available at this time.
Zombie	When the parent of a process terminates before it executes the exit call, it becomes a zombie process. The process finishes and finds that the parent isn't waiting. The zombie processes are finished for all practical purposes and don't reside in the memory, but they still have some kernel resources allocated to them and can't be taken out of the system. All zombies (and their live children) are eventually adopted by the granddaddy, the init process, which removes them from the system. In general, any dying process is said to be in the zombie state.

More states



From: *Bulletproof Unix* by Tim Gottleber, 2003

Foreground and background

- When a command is executed from the prompt and runs to completion at which time the prompt returns, it is said to run in the foreground
- When a command is executed from the prompt followed by the token '&' on the command line, the prompt immediately returns while the command is said to run in the background
- Programs that interact with a user should be run in the foreground
- Programs that execute slowly and without intervention belong in the background – so other work can get done!
  - e.g., daemons (background processes for system administration)

User control of process state

- Terminate a foreground process with ctrl-C
- Send running foreground process to background by ctrl-Z

```
-bash-4.2$ find / *.txt > /dev/null 2> /dev/null
    ← entered ctrl-Z here
[1]+  Stopped  find / *.txt > /dev/null 2> /dev/null
-bash-4.2$  ← can execute more commands while find works
```

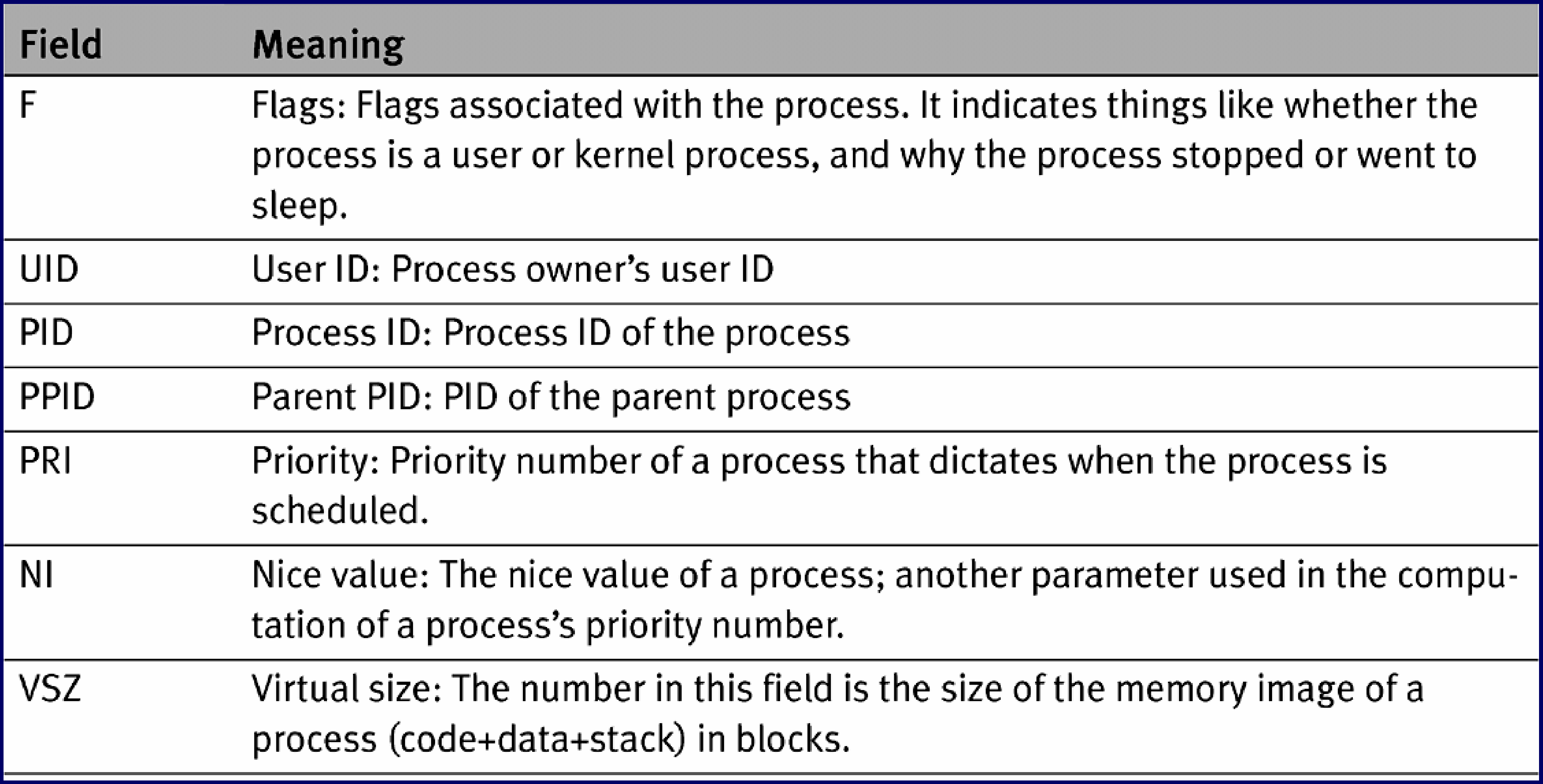
  - If enter `fg 1` now, job 1 will execute in foreground again
- Use ps to find PIDs of running processes

```
-bash-4.2$ ps
  PID TTY          TIME CMD
20637 pts/4    00:00:00 bash
21581 pts/4    00:00:02 find
21632 pts/4    00:00:00 ps
```

- Terminate a background process with kill command

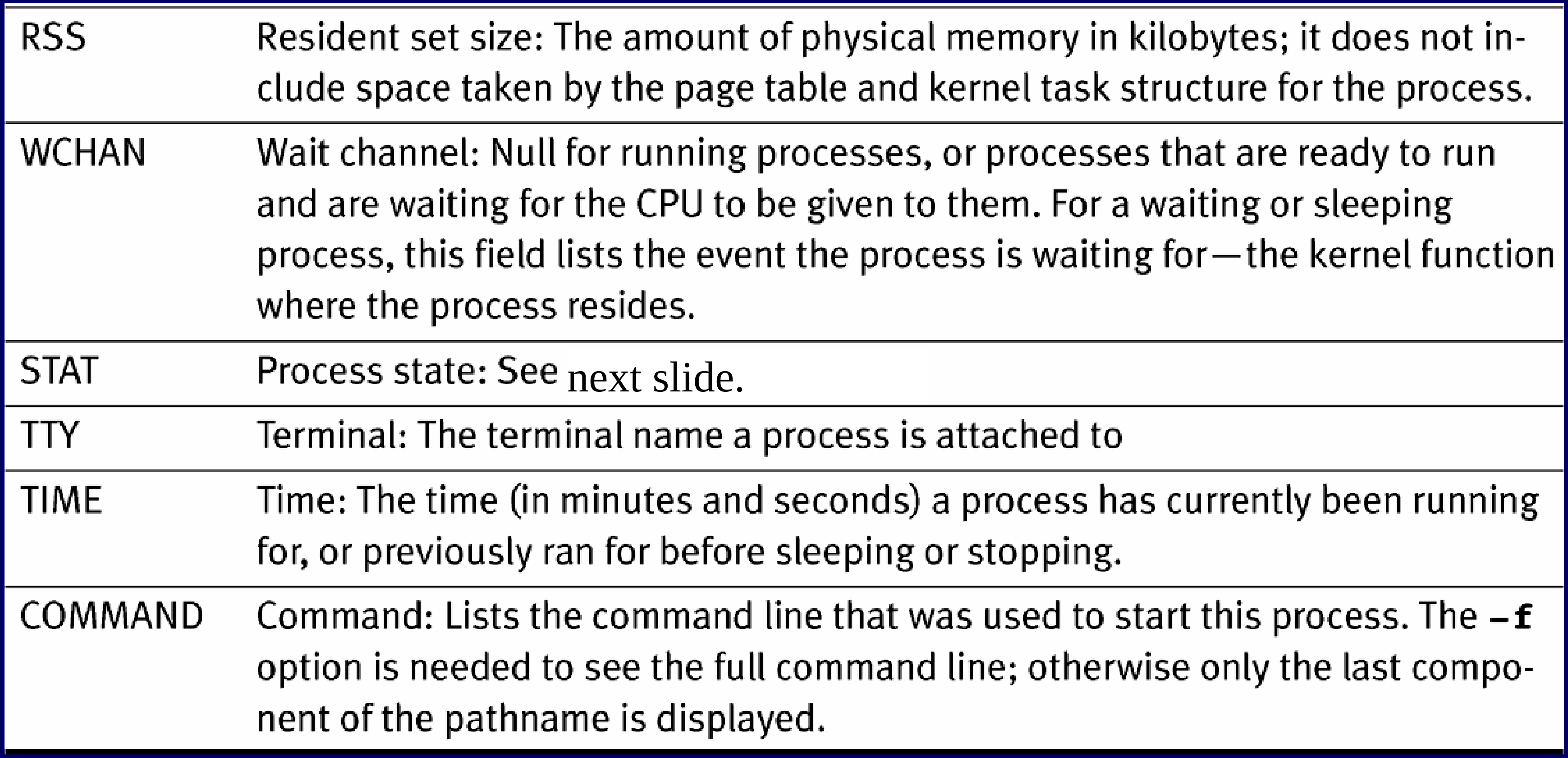
```
bash-4.2$ kill -9 21581     ← -9 is the "sure kill" signal number
-[1]+  Killed     find / *.txt > /dev/null 2> /dev/null
```

Fields of `ps -l` output (cont. next slide)



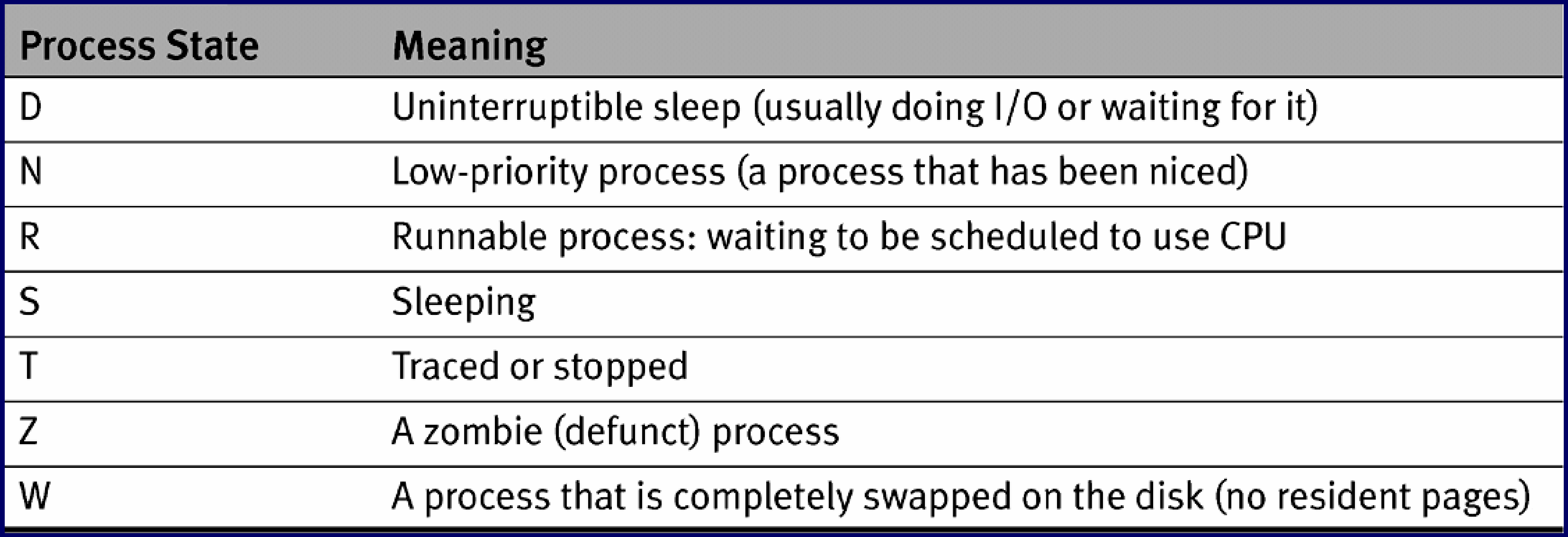
Field	Meaning
F	Flags: Flags associated with the process. It indicates things like whether the process is a user or kernel process, and why the process stopped or went to sleep.
UID	User ID: Process owner's user ID
PID	Process ID: Process ID of the process
PPID	Parent PID: PID of the parent process
PRI	Priority: Priority number of a process that dictates when the process is scheduled.
NI	Nice value: The nice value of a process; another parameter used in the computation of a process's priority number.
VSZ	Virtual size: The number in this field is the size of the memory image of a process (code+data+stack) in blocks.

Fields of `ps -l` output (cont.)



RSS	Resident set size: The amount of physical memory in kilobytes; it does not include space taken by the page table and kernel task structure for the process.
WCHAN	Wait channel: Null for running processes, or processes that are ready to run and are waiting for the CPU to be given to them. For a waiting or sleeping process, this field lists the event the process is waiting for—the kernel function where the process resides.
STAT	Process state: See next slide.
TTY	Terminal: The terminal name a process is attached to
TIME	Time: The time (in minutes and seconds) a process has currently been running for, or previously ran for before sleeping or stopping.
COMMAND	Command: Lists the command line that was used to start this process. The **-f** option is needed to see the full command line; otherwise only the last component of the pathname is displayed.

Process state abbreviations



Process State	Meaning
D	Uninterruptible sleep (usually doing I/O or waiting for it)
N	Low-priority process (a process that has been niced)
R	Runnable process: waiting to be scheduled to use CPU
S	Sleeping
T	Traced or stopped
Z	A zombie (defunct) process
W	A process that is completely swapped on the disk (no resident pages)