UNIVERSITY of WEST FLORIDA

COP4634: Systems & Networks I

Processes

Programs

- Compiled, executable code
- Stored on disk
- **Passive** entity
- Doesn't **do** anything
- Read from disk
- Written to memory
- Execution begun

Done by loader

no longer program

now a process

Process Concept

- Definition: A process (= job) is an instance of a program in memory whose instructions are executed sequentially by a CPU.

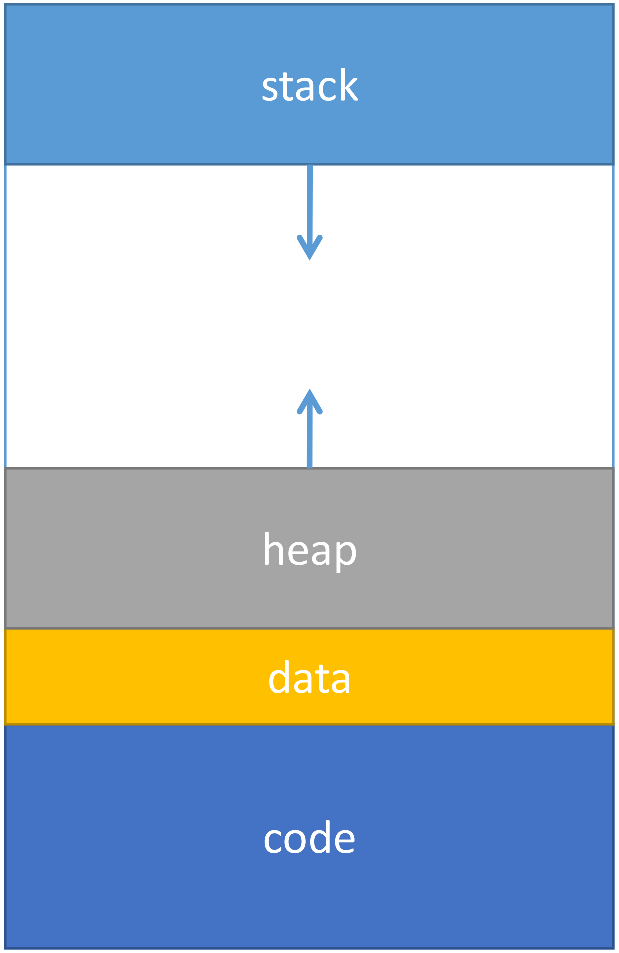
- Modern operating system (OS) can execute multiple processes concurrently.
  - no real, only virtual concurrent execution on a single core, single CPU
  - OS switches fast between processes giving each process a chance to run on the CPU

- OS maintains a process control block for each process
  - a data structure maintains information about the process

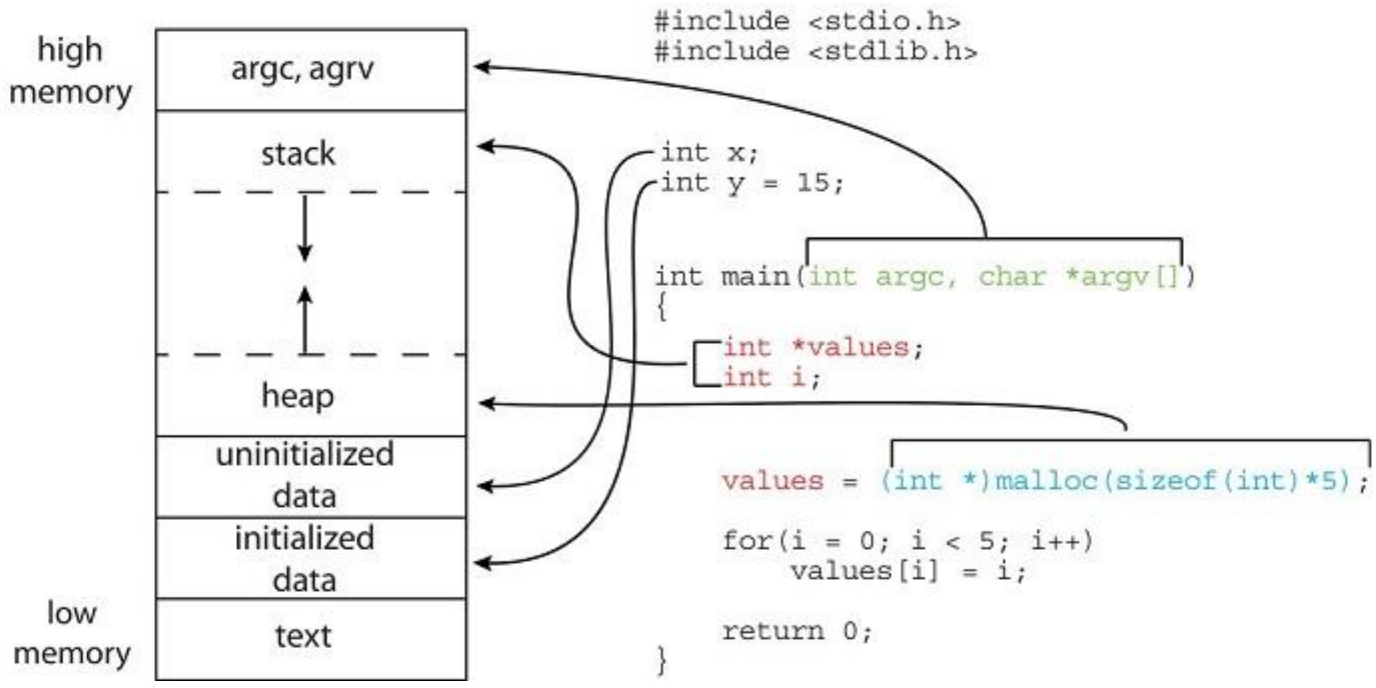
Process Layout in Memory

Process consists of

- code: a sequence of instructions
- data: global variables
- stack: local variables, function parameters generated during execution
- heap: dynamic memory allocated during run-time



Process Layout Illustration



Process Control Block (PCB)

OS maintains a PCB for each process to keep track of a process in memory:

- Process ID,
- Process state (e.g. running, ready, waiting, …),
- Program counter (address of next instruction),
- CPU registers,
- CPU scheduling information,
- File management (list of open files, working directory, …),
- I/O status information,
- Memory management information (pointers to text, data, stack).

States in Process Life Cycle

Process assume different states during execution life cycle:

New:

Running:

Waiting:

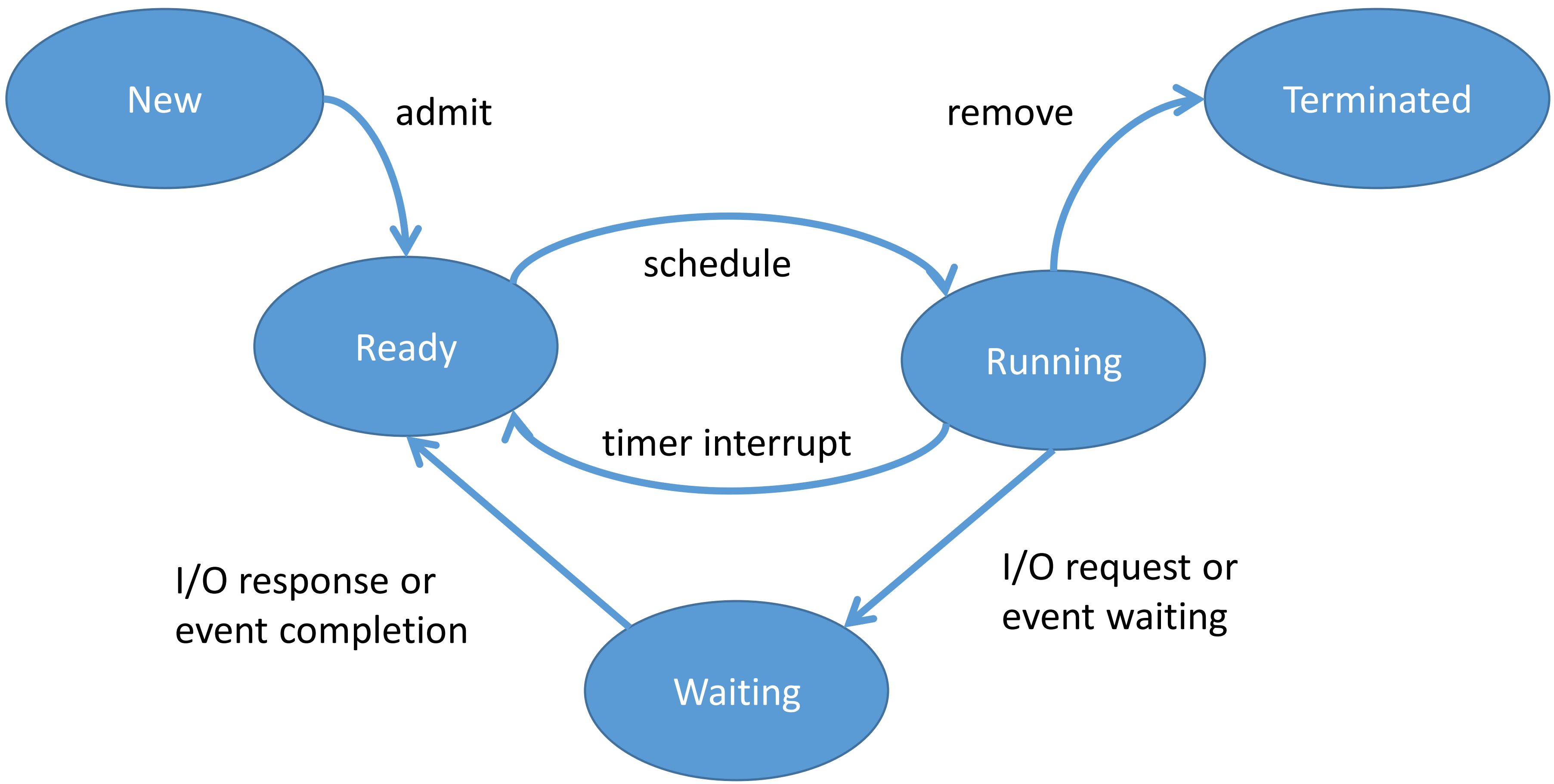
(e.g. an IO device returning some results)

Ready:

Terminated:

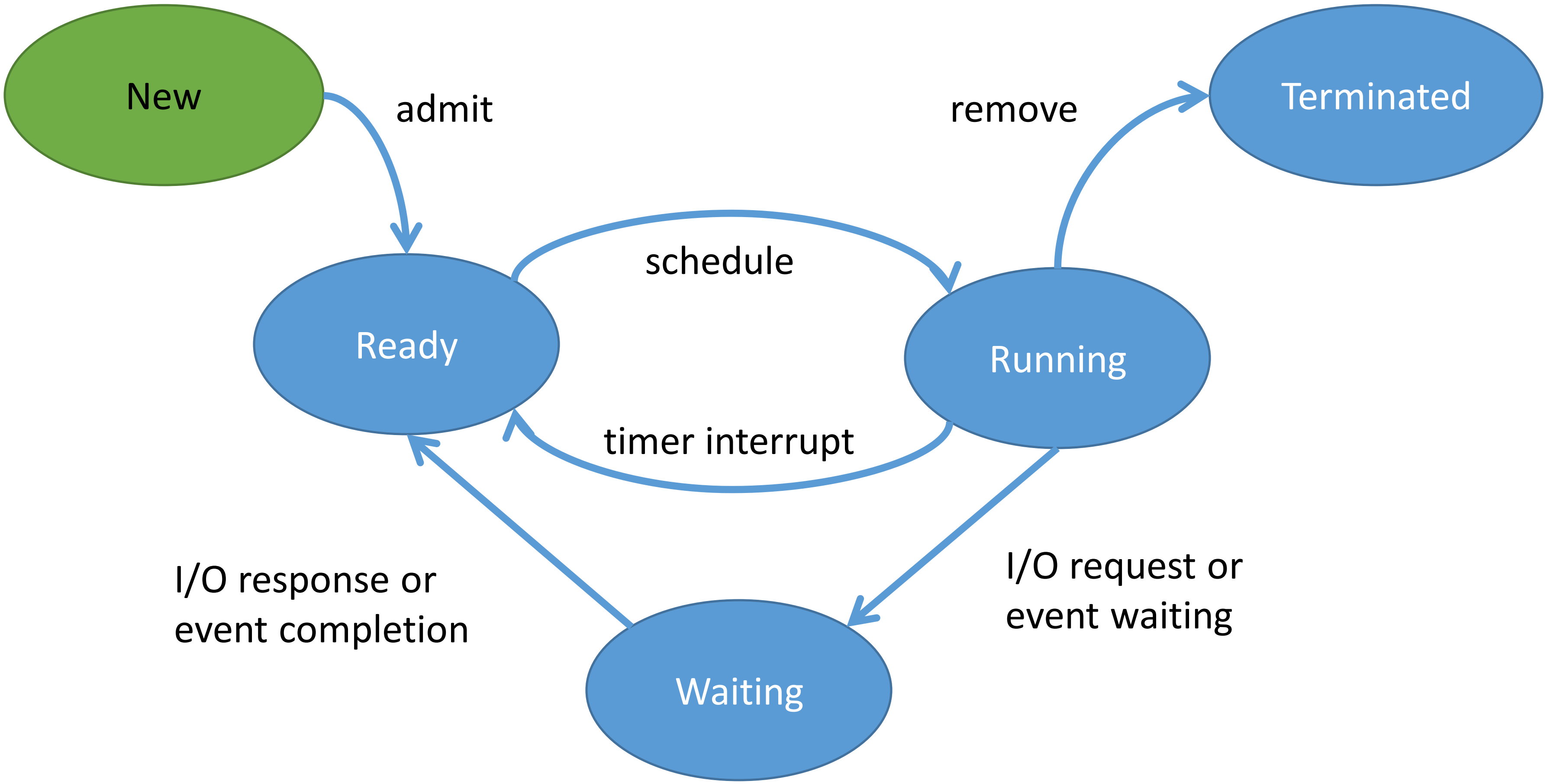
Process State Diagram (Illustrated)

During execution process is in one of five different states.



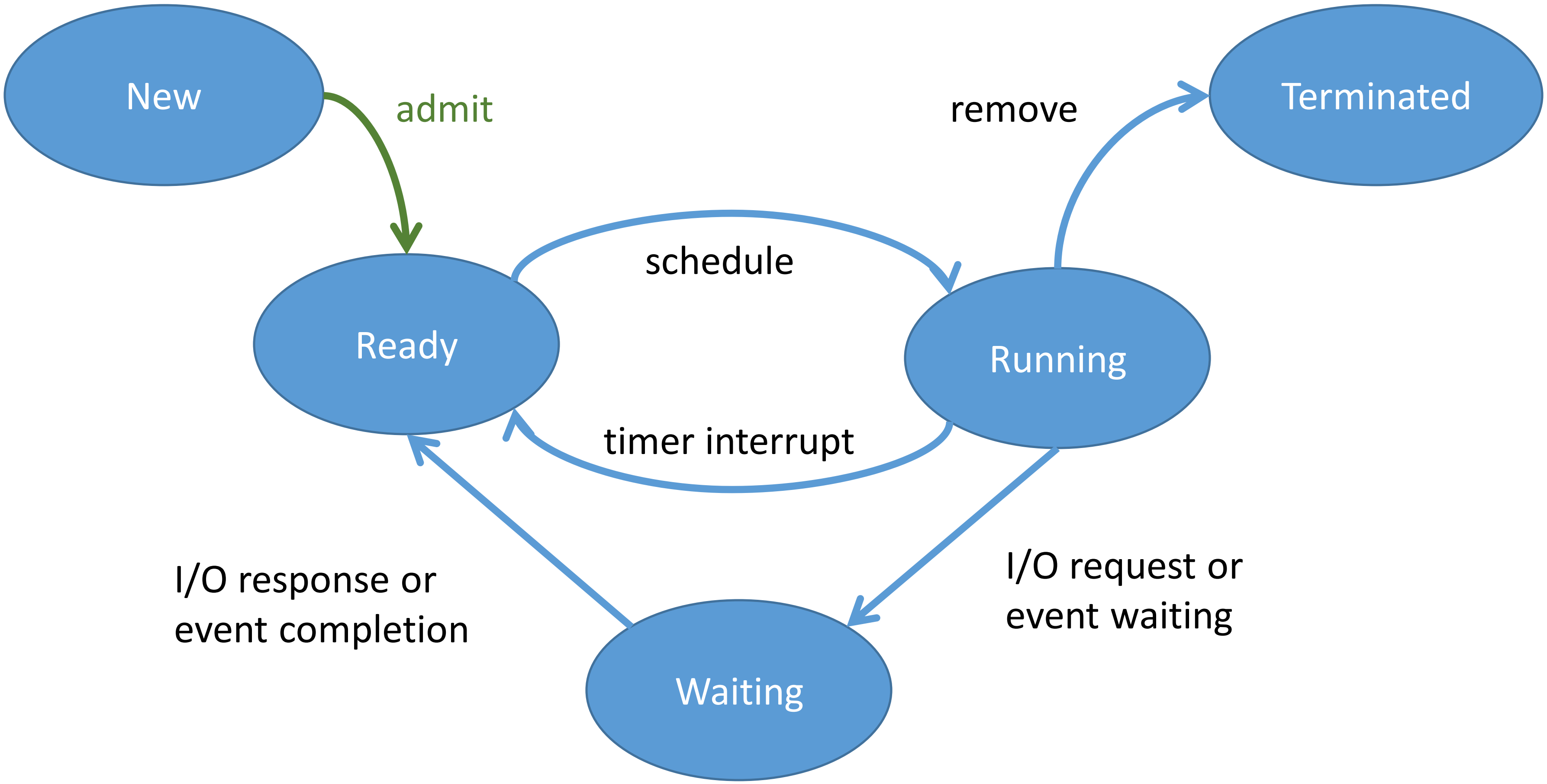
Process State Diagram (Illustrated)

Kernel is aware of another process requesting to be started.
No memory has been allocated.



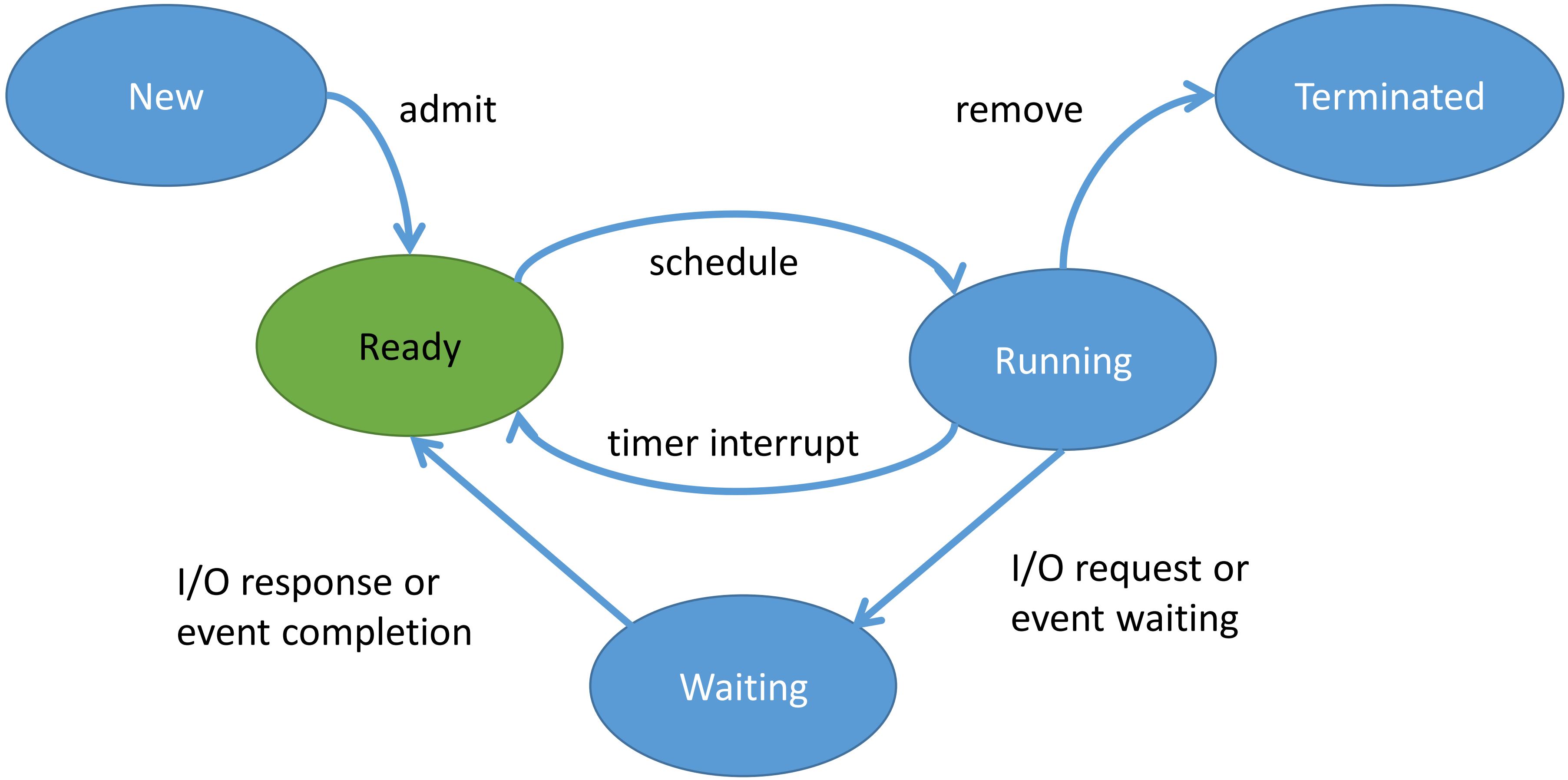
Process State Diagram (Illustrated)

PCB is allocated and initialized. Memory for the new process is allocated. Process is ready for execution.



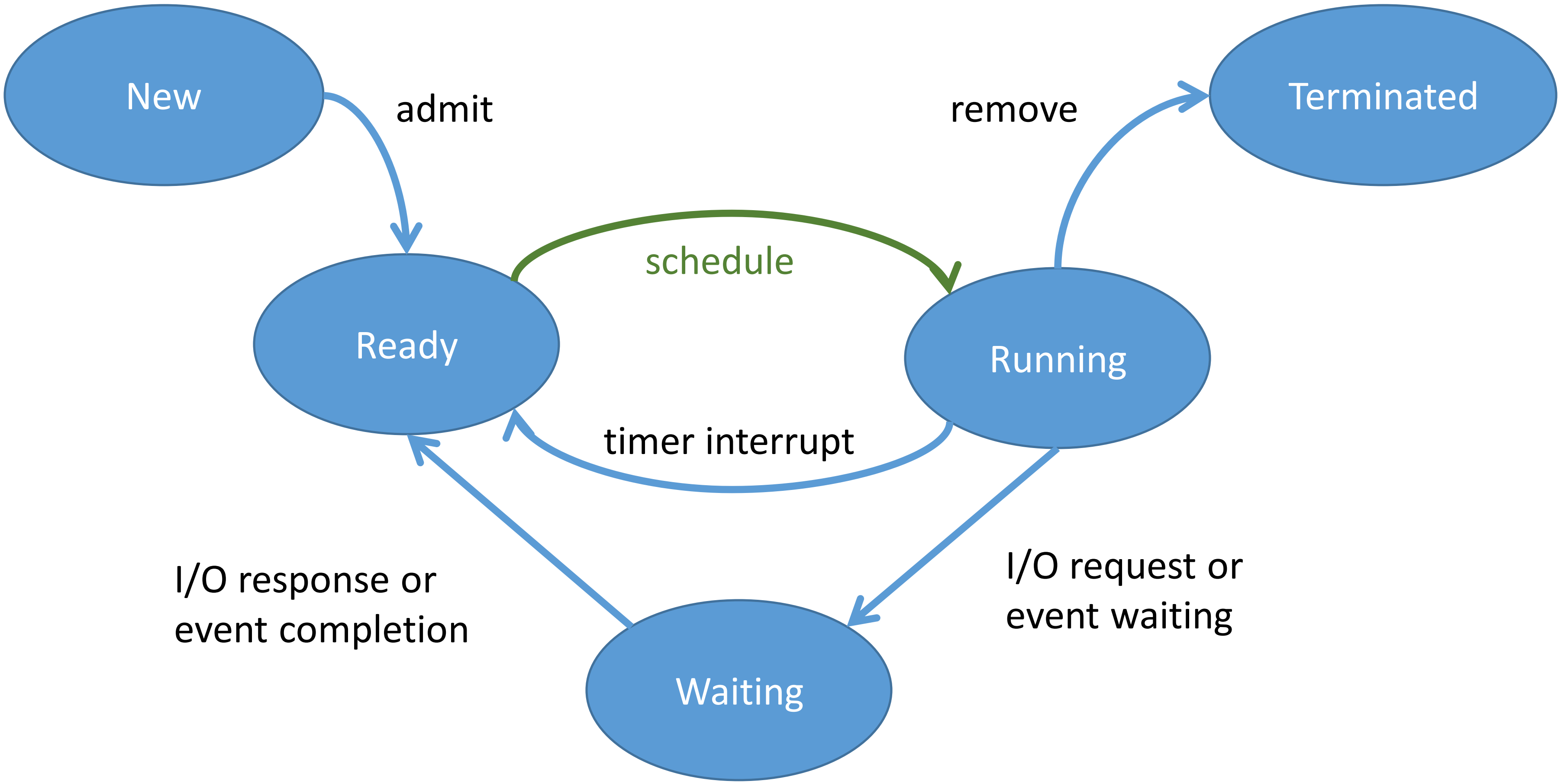
Process State Diagram (Illustrated)

Process is waiting, ready for access to CPU.
OS CPU scheduler may select process for execution.



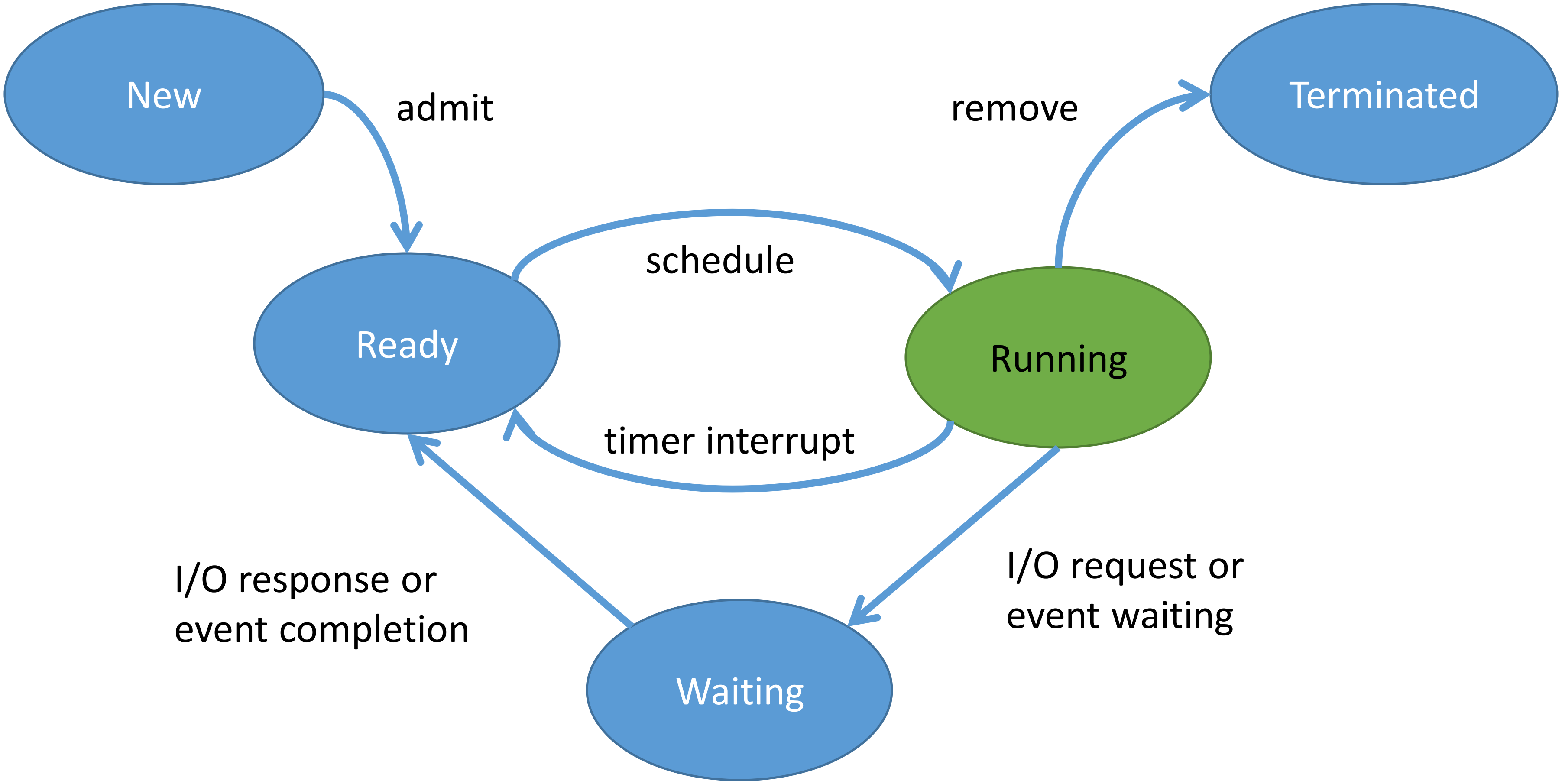
Process State Diagram (Illustrated)

Process is selected for execution by scheduler.
Alarm interrupt is set for *now + quantum.*
Process is “placed” on CPU; state changes to running.



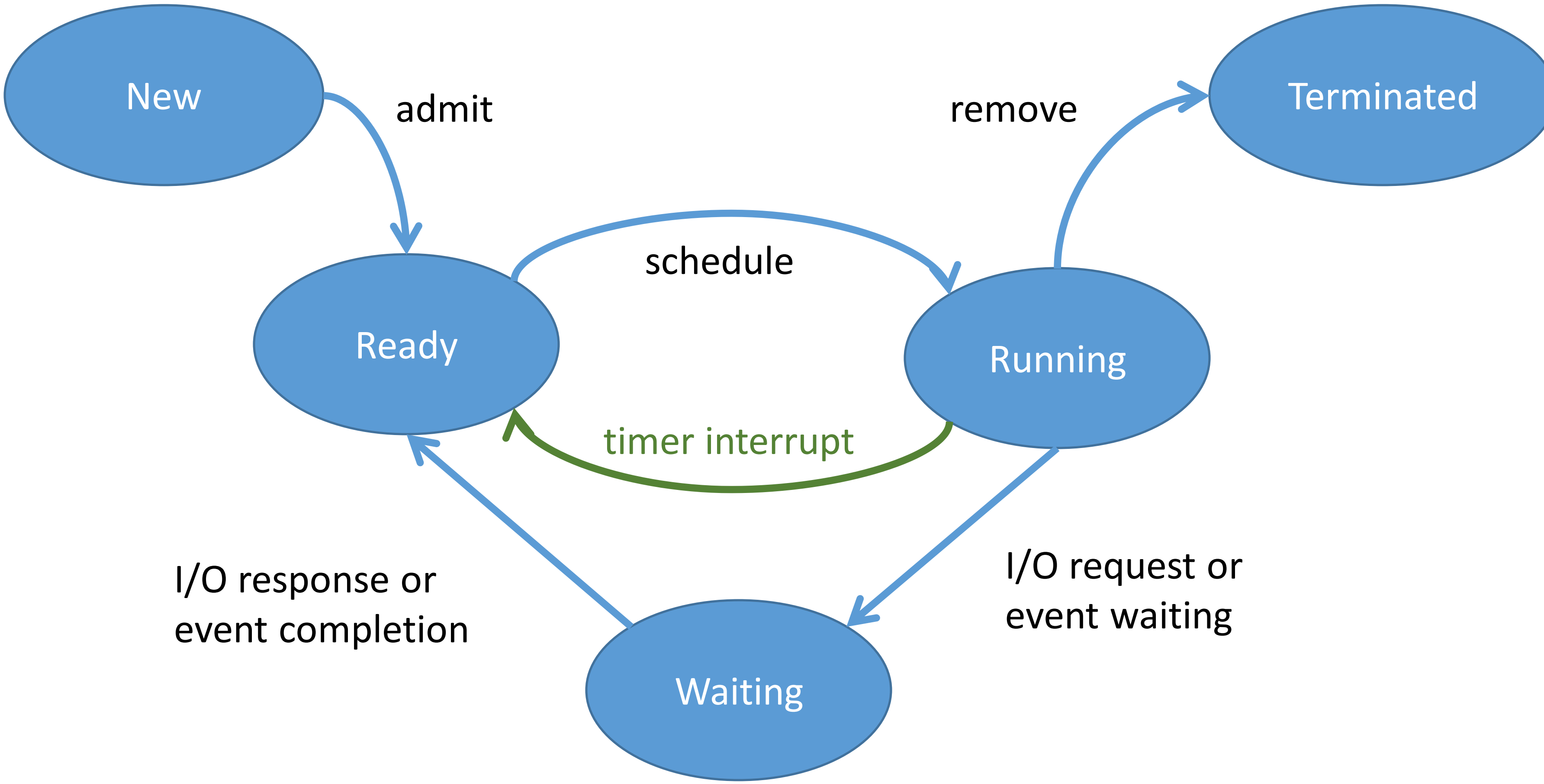
Process State Diagram (Illustrated)

CPU fetches and executes process instructions. Process may loose CPU access in three different ways.



Process State Diagram (Illustrated)

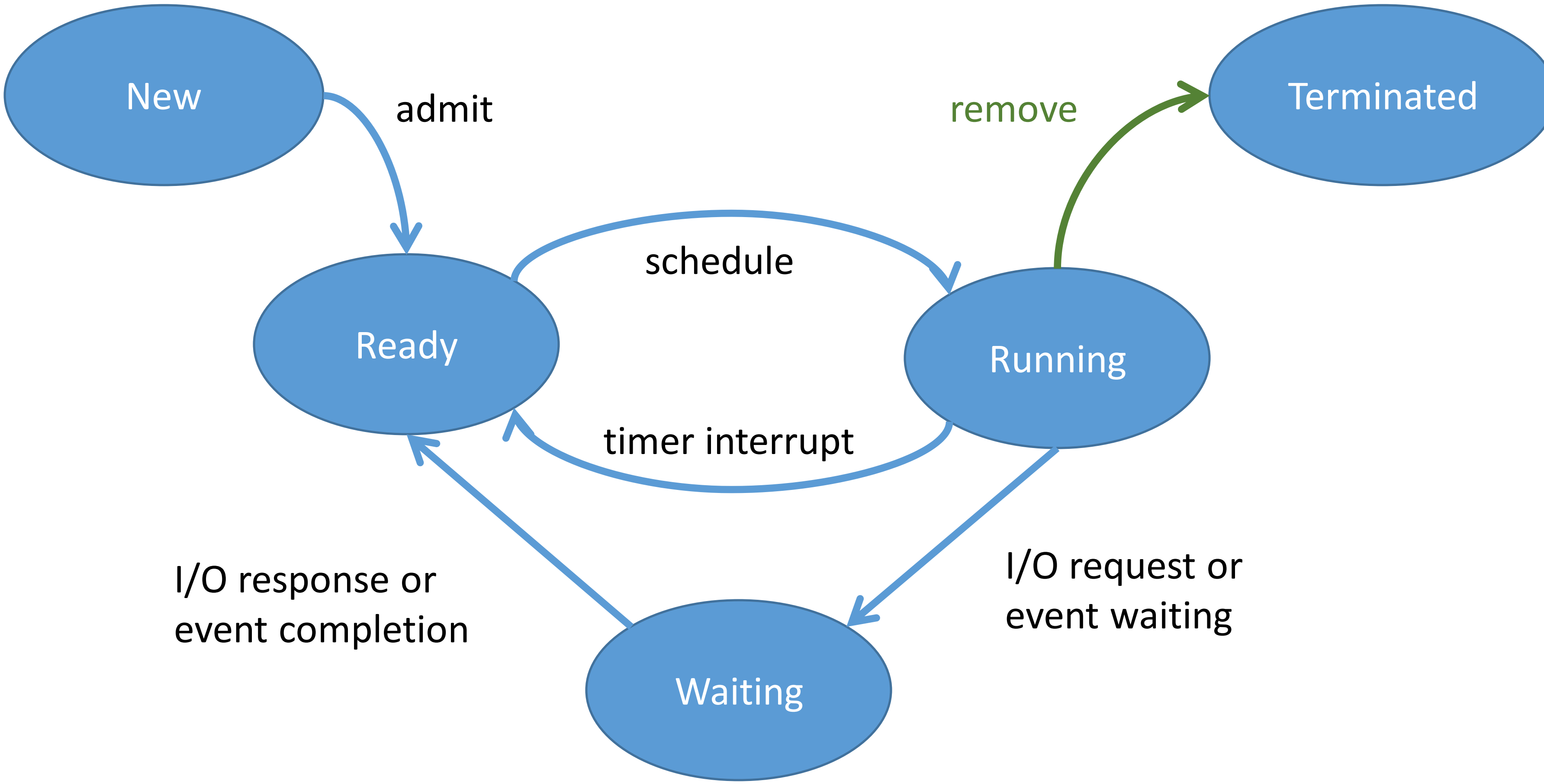
Quantum expires – Interrupt triggers kernel to resume control.
Process is moved to *ready list* and next process is *scheduled.*



Process State Diagram (Illustrated)

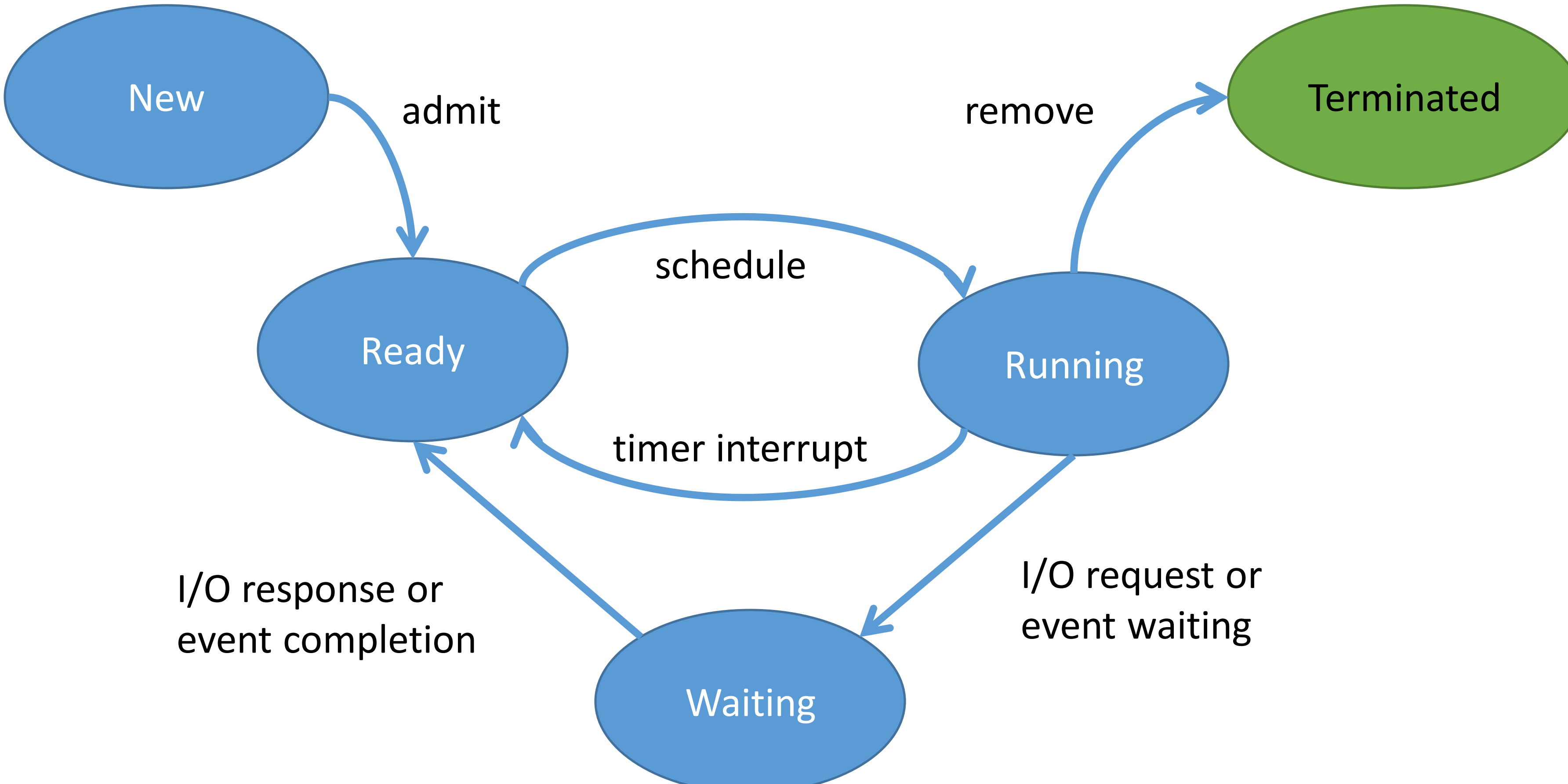
Process exits – return from `main()`, calls `exit()`, exception.

Still a process – resources remain allocated until deleted by OS..



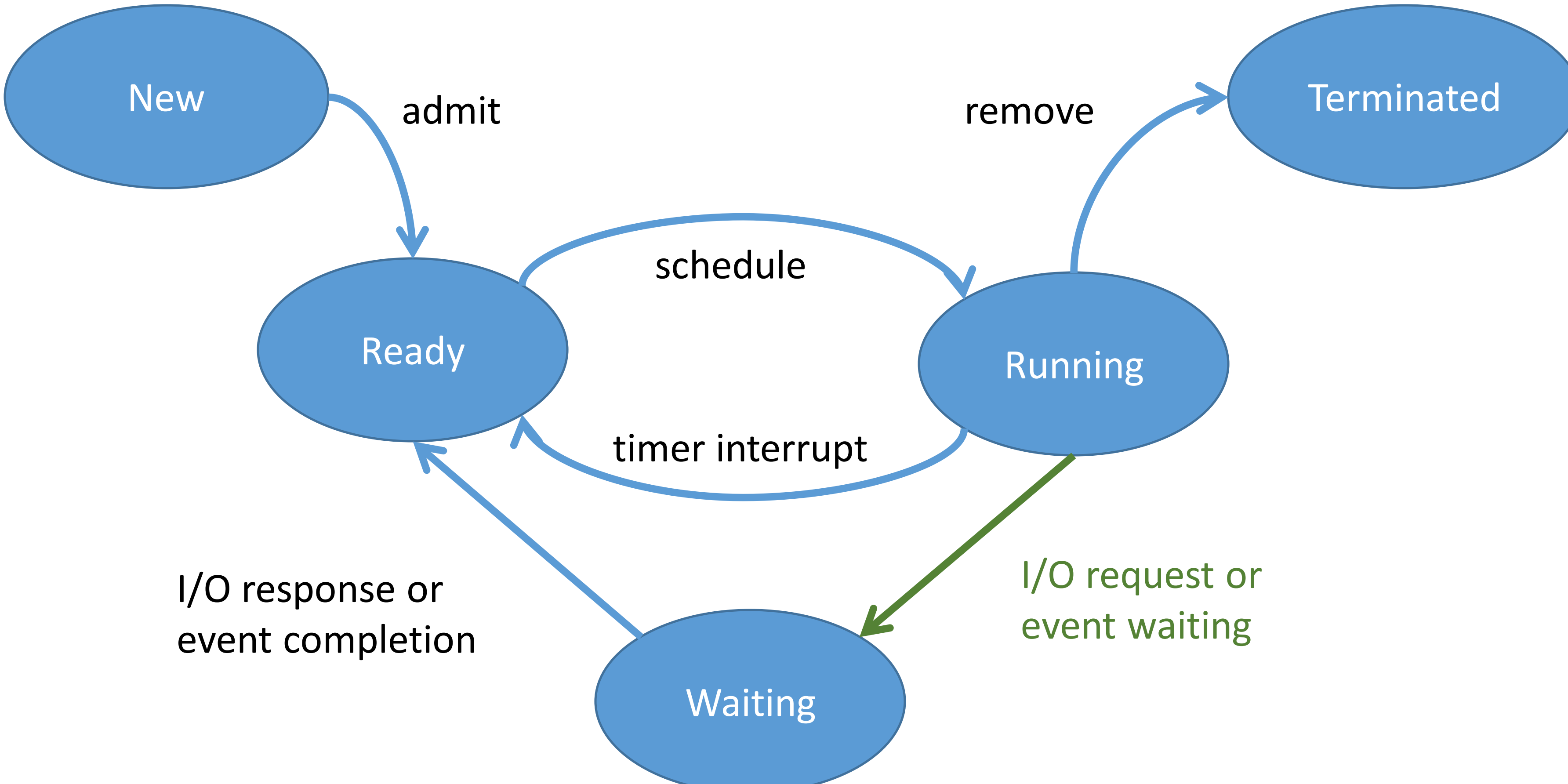
Process State Diagram (Illustrated)

Kernel reclaims resources and return value is given to parent.
Could become a *zombie* process.



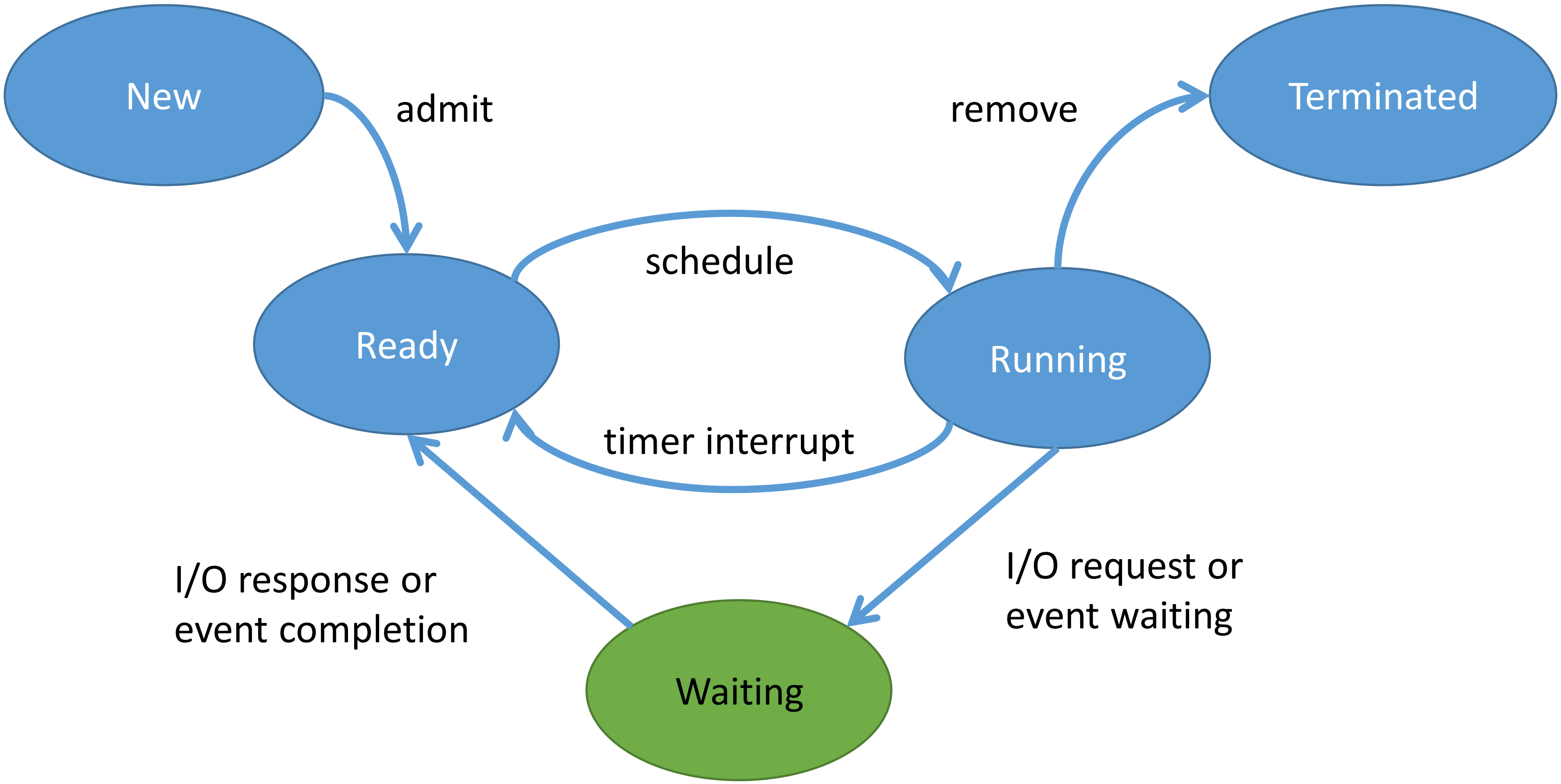
Process State Diagram (Illustrated)

Process initiates input or output or could *wait()* for other processes. Process started something that will take a while.



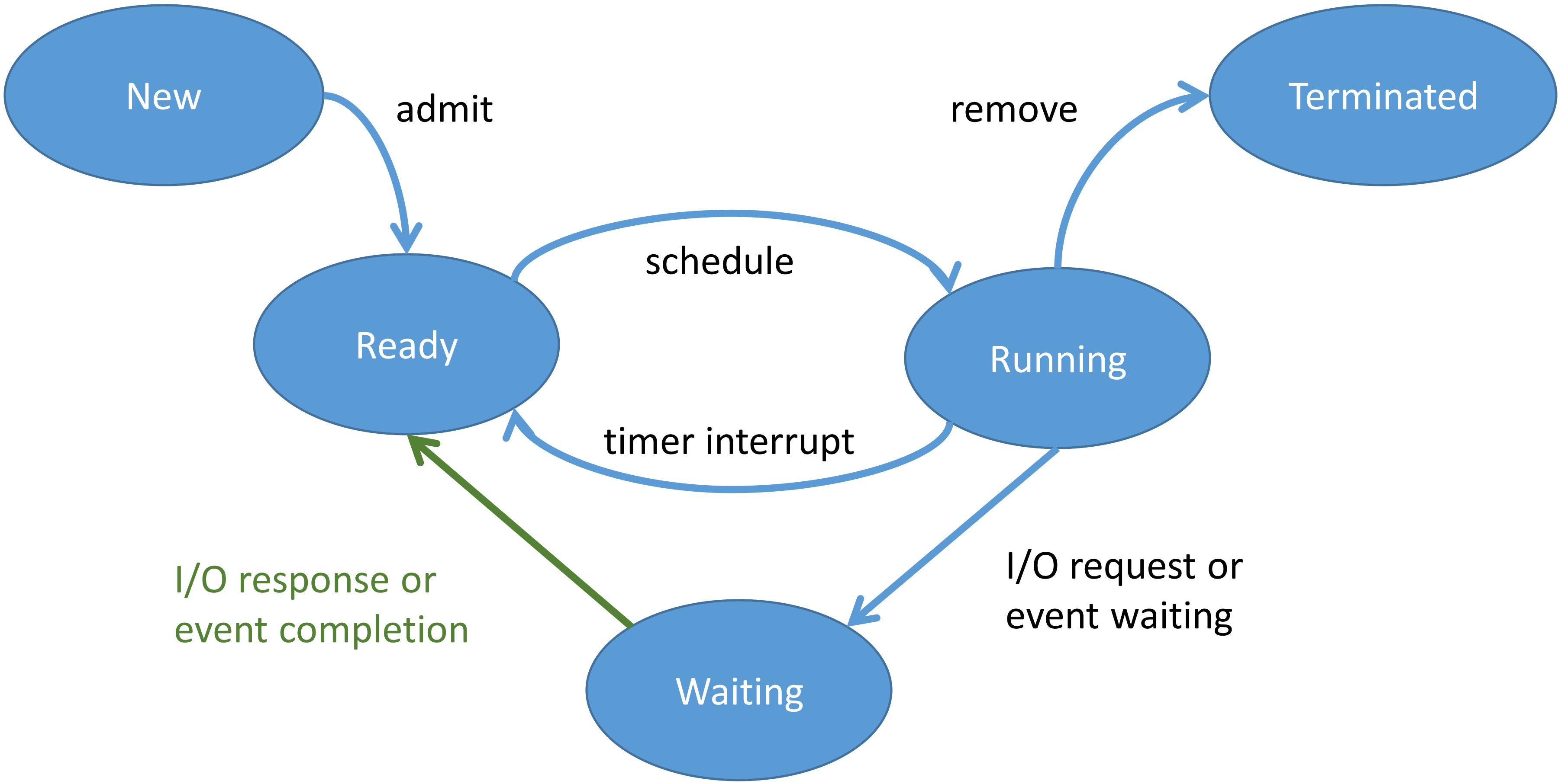
Process State Diagram (Illustrated)

Process waits for input or output to complete or for another process to wake it up. Process is not ready to continue execution.



Process State Diagram (Illustrated)

Input/Output or `wait()` is completed; process is ready to be selected for execution.



Inside the CPU

- General purpose registers (R0, R1, …)
- Special purpose registers
  - IP/PC: instruction pointer/program counter
  - IR: instruction register
  - SP: stack pointer
- More stuff later

Fetch/Execute Cycle

1. Fetch instruction at address PC into IR
2. Increment PC (by 1, 2, 4, ?)
3. Decode instruction in IR
4. Execute instruction in IR
5. *Check for interrupts*
6. Go back to step 1

Runtime Stack, 1

- SP has address of top of runtime stack
- Runtime stack contains “activation records”
- Stack grows down
- Call to function:
  - creates activation record for function
  - pushes AR onto stack
  - jumps to code for function

Activation Record

Contains

- parameters
- local variables
- return value
- return address
- other items

```
int add(int x, int y){
   int z;
   z = x + y;
   return z;
}

int main(){
   int val;
   val = add(2,3);
   printf("%d\n",val);
   return 0;
}
```

Function Calls

1. Push arguments onto stack
2. Push local vars onto stack
3. Push return address onto stack
4. Push return value onto stack

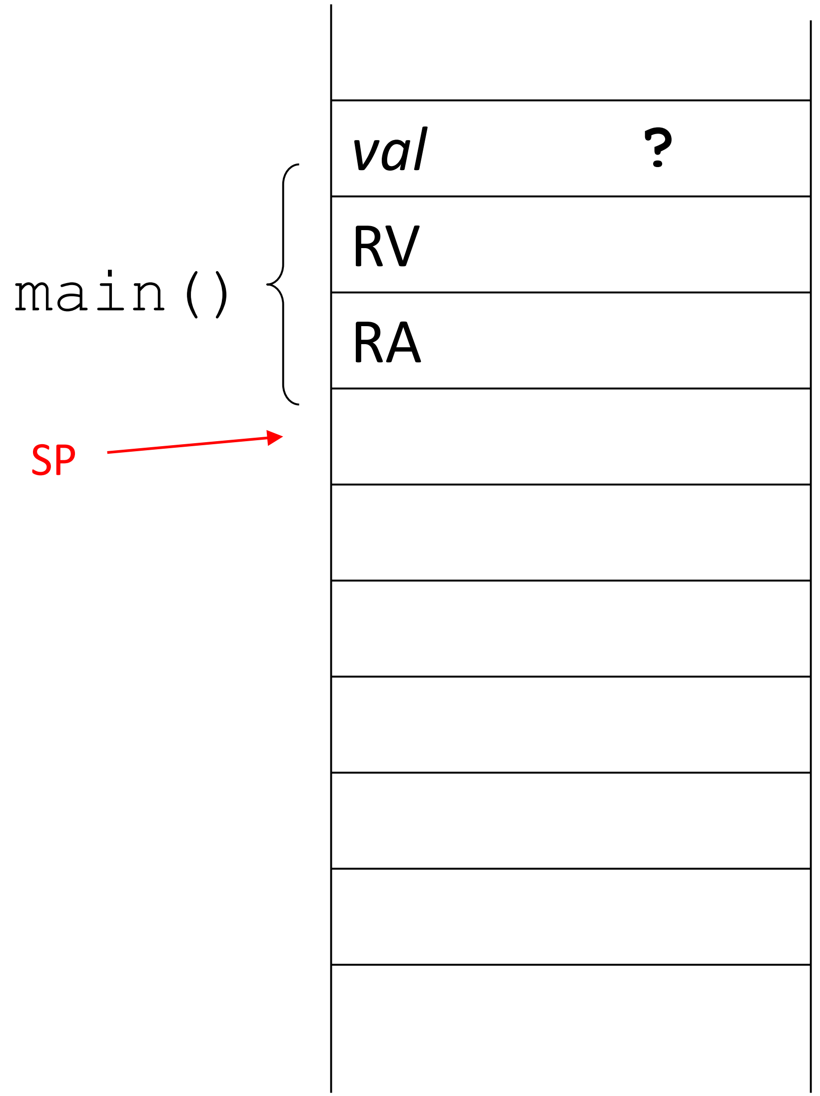
   (created activation record)
5. Jump to user-defined code

Limited in what it can do

Only access to user-mode instructions

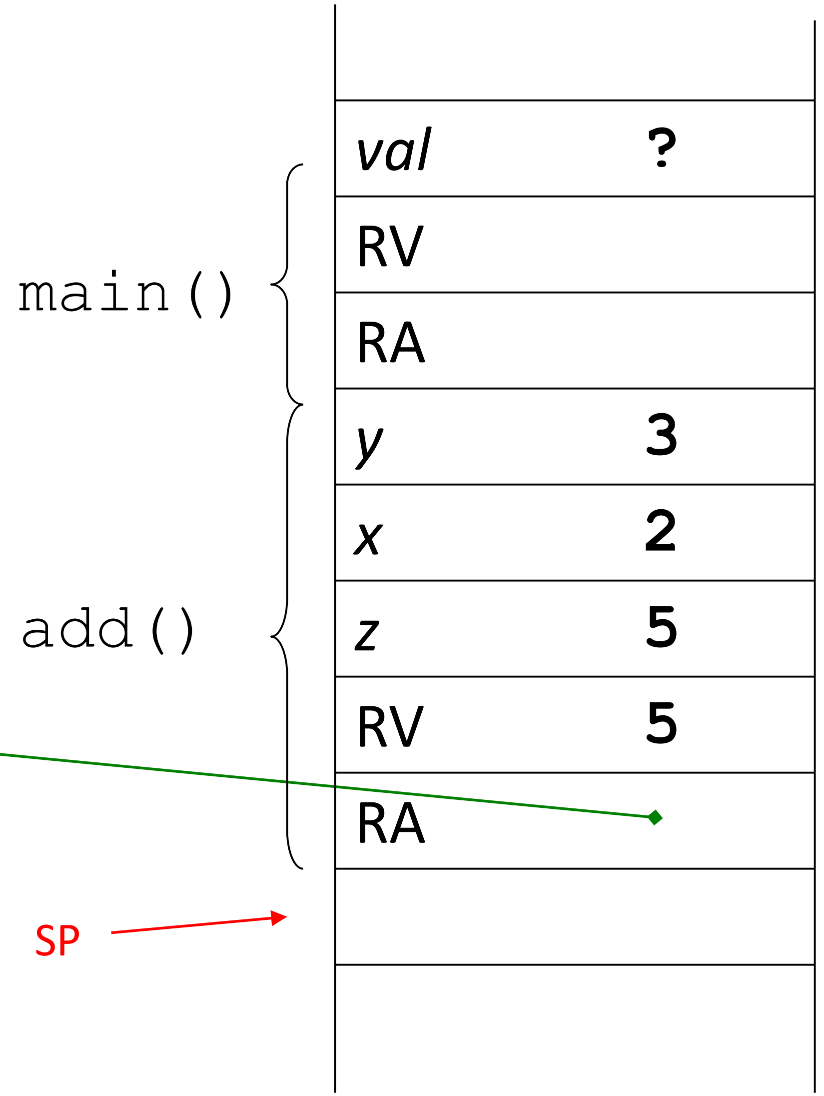
```c
int add(int x, int y){
   int z;
   z = x + y;
   return z;
}
int main(){
   int val;
   val = add(2,3);
   printf("%d\n",val);
}
```

PC



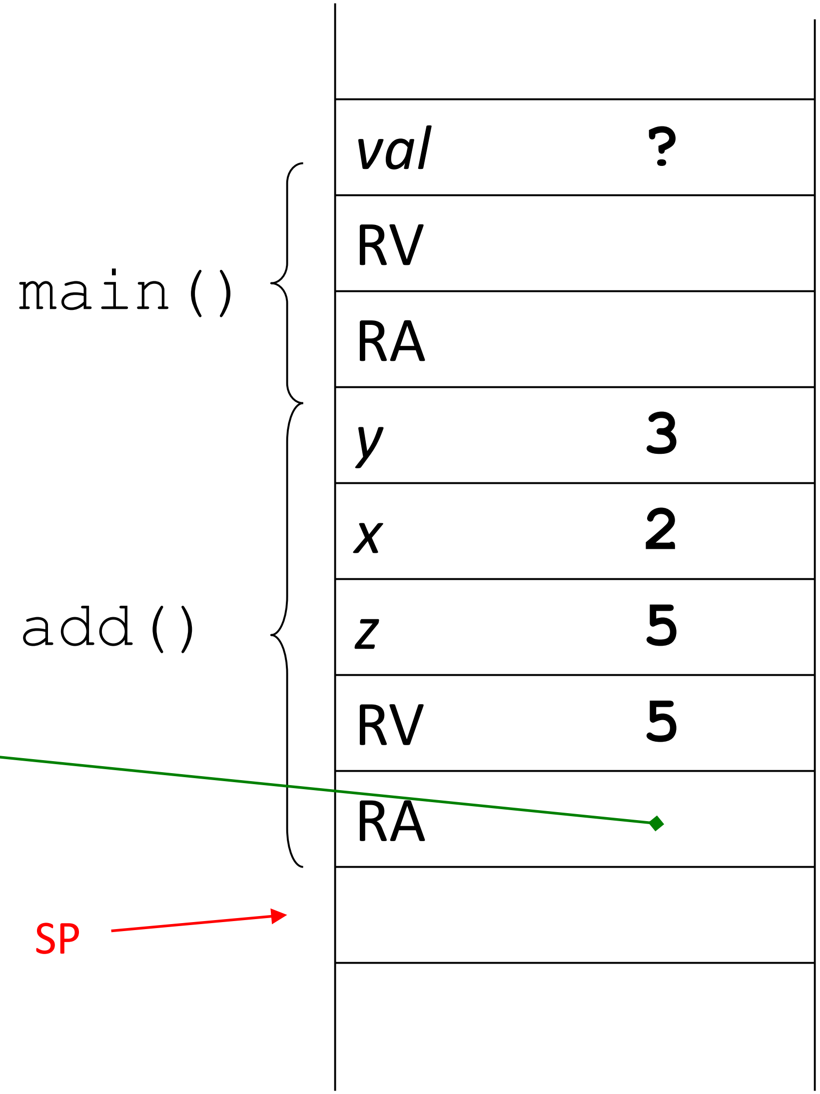
```
int add(int x, int y){
   int z;
   z = x + y;
   return z;
}
int main(){
   int val;
   val = add(2,3);
   printf("%d\n",val);
}
```

PC



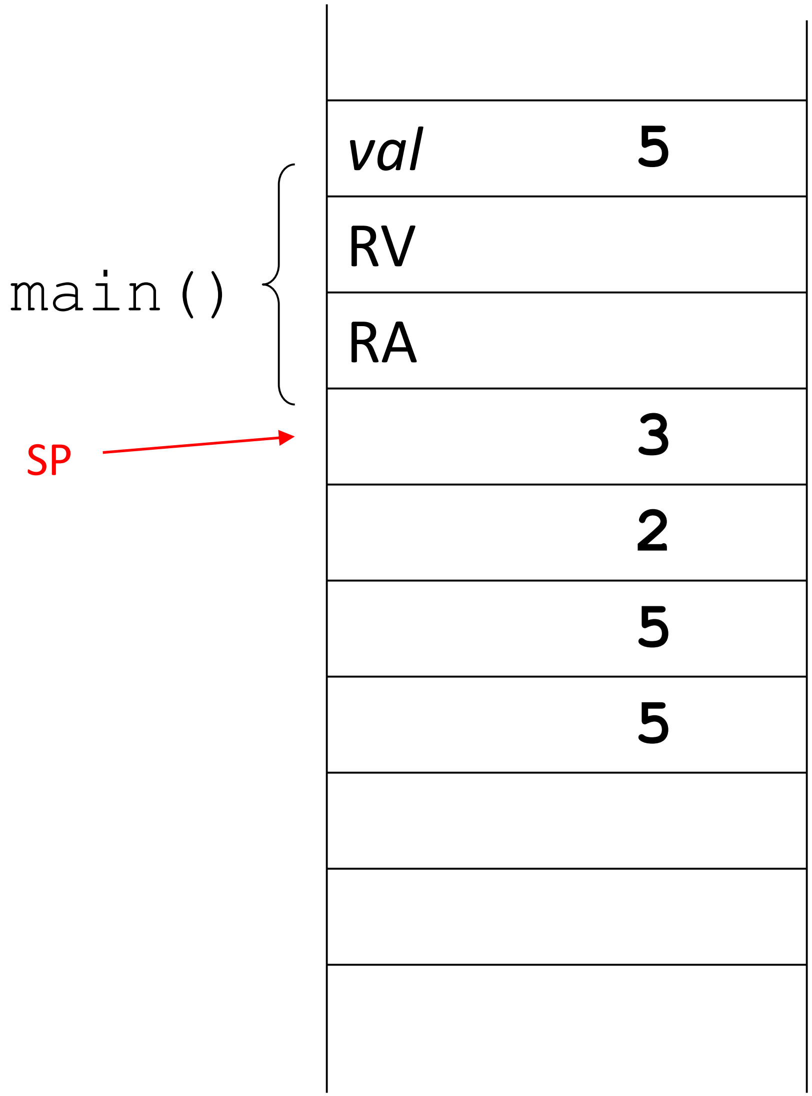
```c
int add(int x, int y){
    int z;
    z = x + y;
    return z;
}
int main(){
    int val;
    val = add(2,3);
    printf("%d\n",val);
}
```

PC



```
int add(int x, int y){
   int z;
   z = x + y;
   return z;
}
int main(){
   int val;
   val = add(2,3);
   printf("%d\n",val);
}
```

PC



Execution Modes

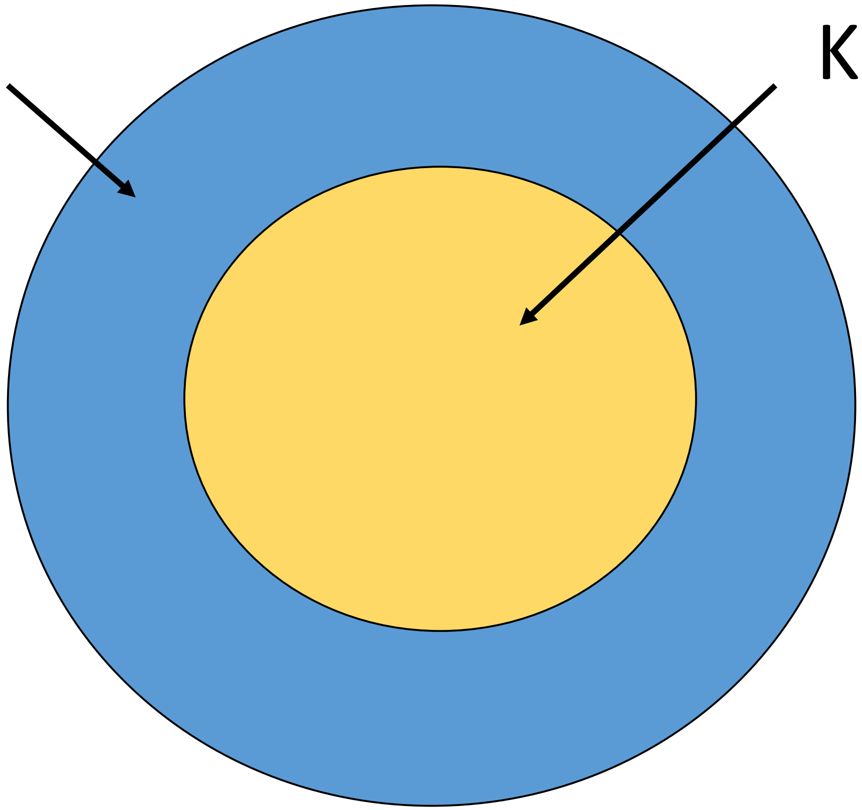
User-Mode

`add`

`mult`

`load`

`stor`



Kernel-Mode

`in`

`out`

`port`

`cmod`

CPU

CPU has execution **mode** bit

- 0 – CPU is currently in kernel-mode
- 1 – CPU is currently in user-mode

`cmod` instruction changes mode

from kernel to user

No instruction to change

from user to kernel (Why?)

System Calls

1. Push arguments onto stack
2. Push local vars onto stack
3. Push return address onto stack
4. Push return value onto stack

   (created activation record)
5. Switch to kernel mode – How?
6. Jump to system-defined code

Access to all instructions

Transitions to Kernel Mode

1. Software

- System call
- Called a “trap” into the kernel
- Jump to well-known function
- Hardware switches to kernel mode
- Example: call to `printf()` or `read()`

Transitions to Kernel Mode

2. Exception

- Error state or debugging
- Similar to system call without return (error)
- Jump to well-known function
- Hardware switches to kernel mode
- Example: division by zero or segmentation violation
- Result: core dump

Transitions to Kernel Mode

3. Hardware

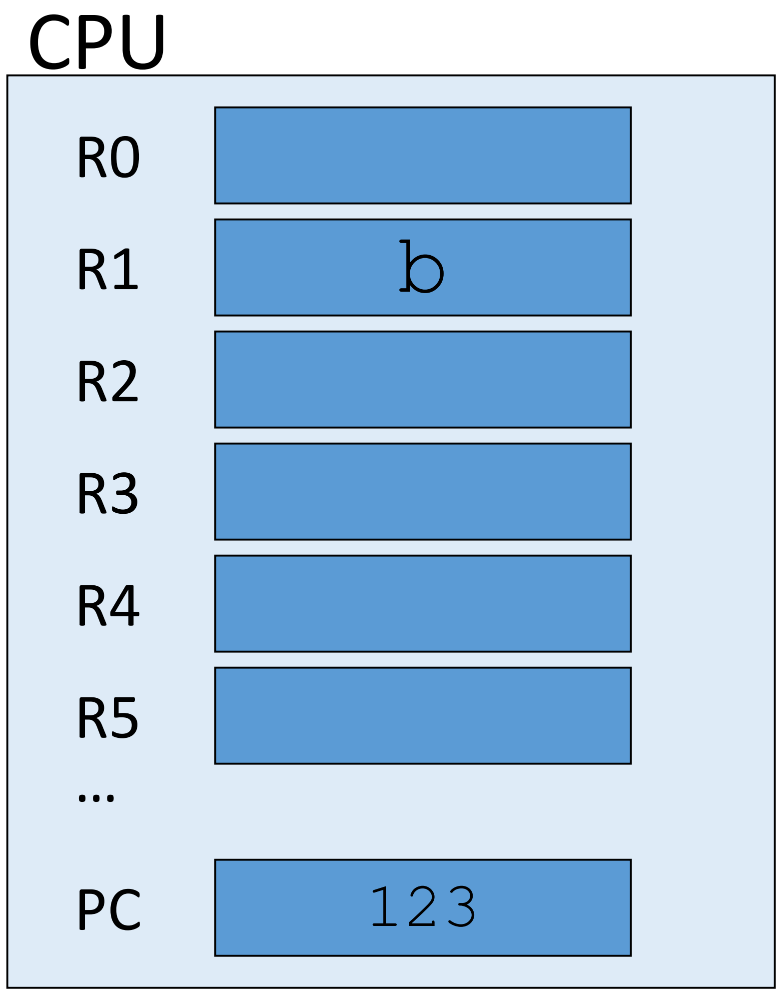
- Called an “interrupt”
- Communication between kernel & devices
- Can occur between any two instructions
- Similar to system call without call
- Jump to well-known function
- Hardware switches to kernel mode
- Example: clock tick or I/O complete

Statements & Instructions

```
int add(int a, int b){
   int c;
   c = a + b;
   return c;
}
```

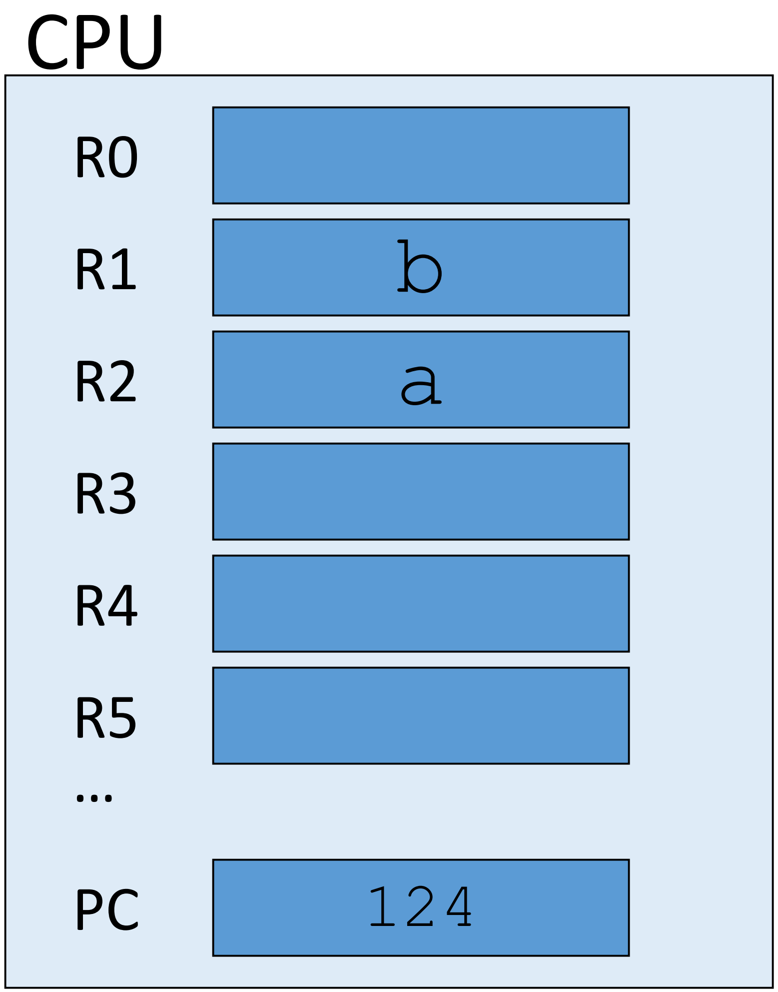
```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

Executing Instructions



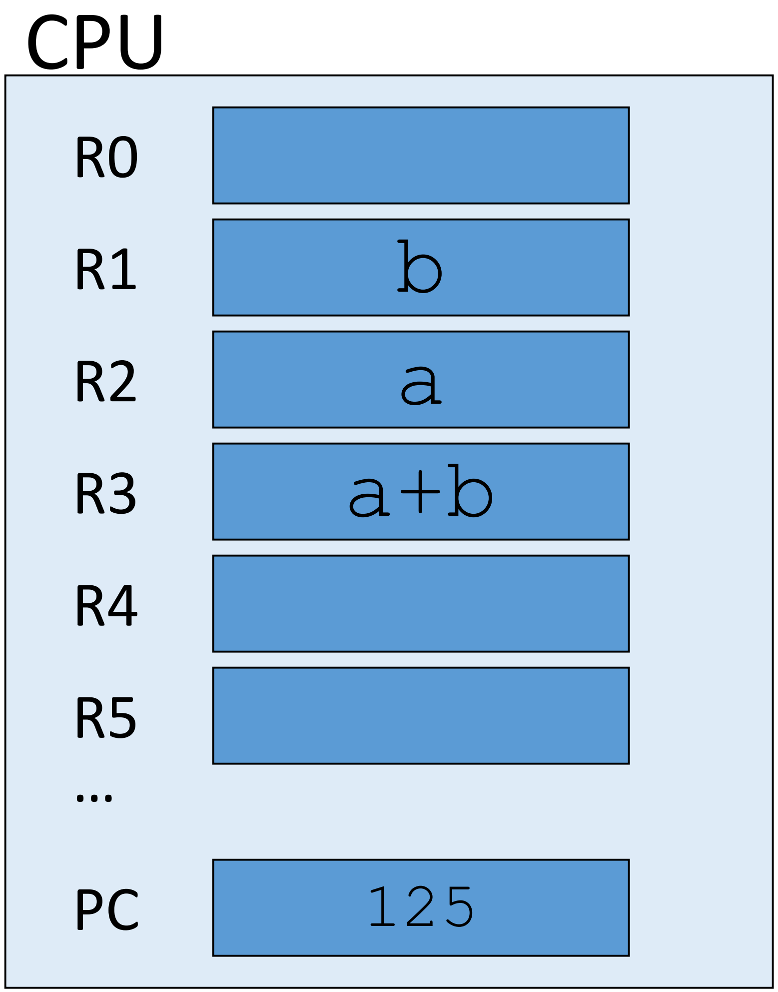

```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

Executing Instructions



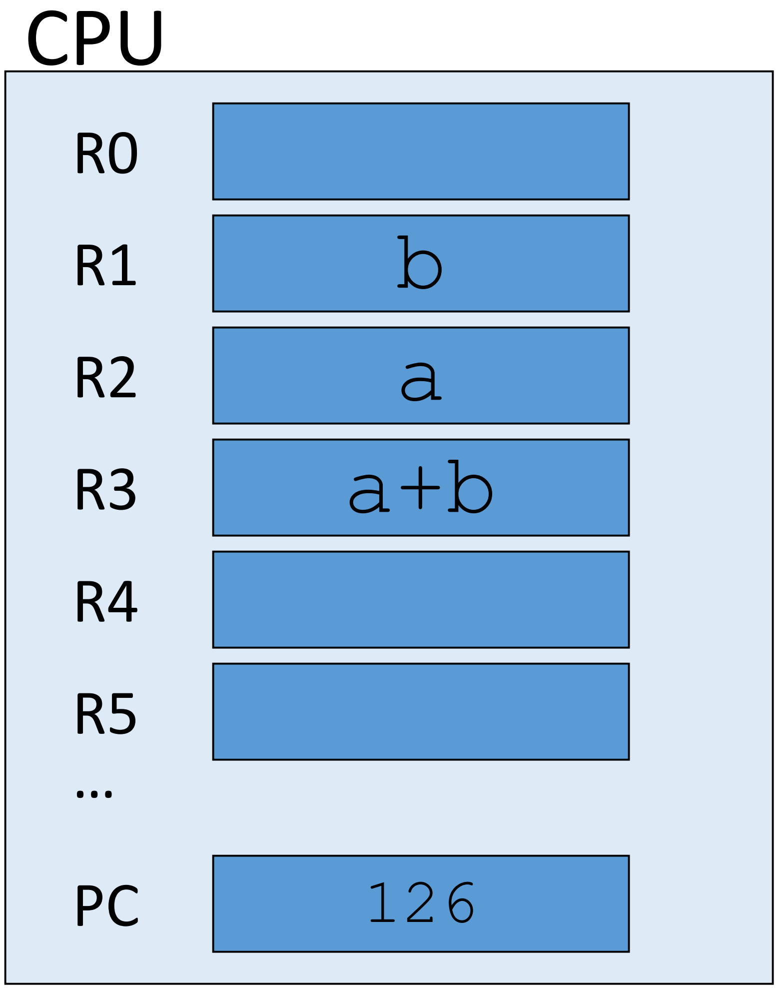

```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

Executing Instructions




```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

Executing Instructions

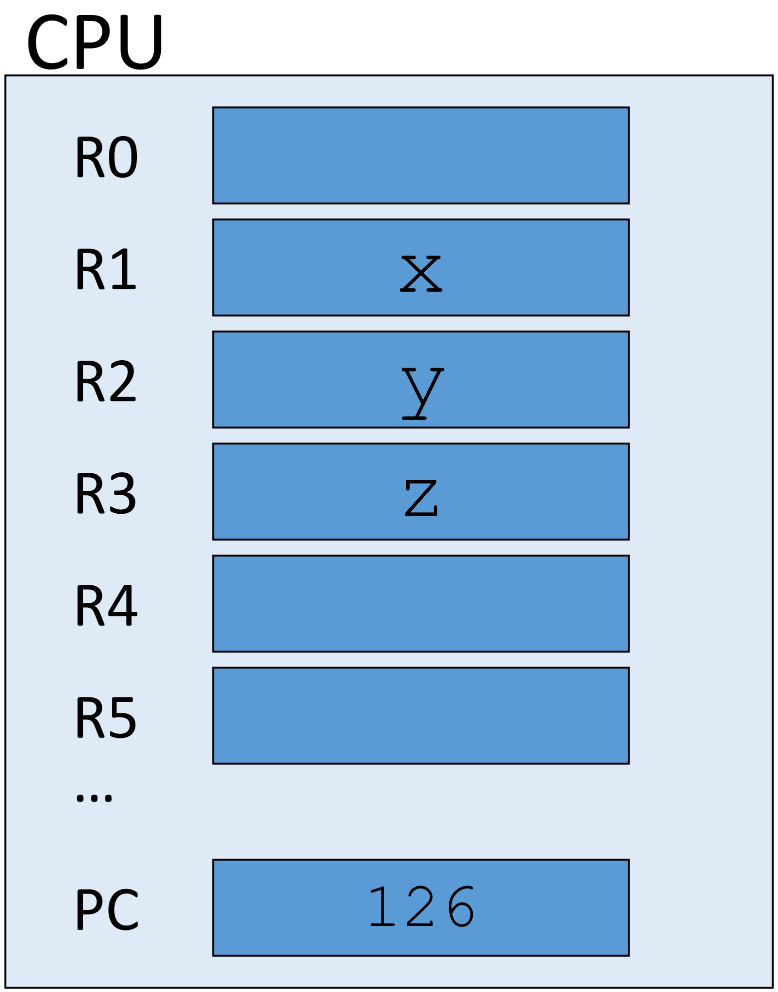



```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

Quantum Expires

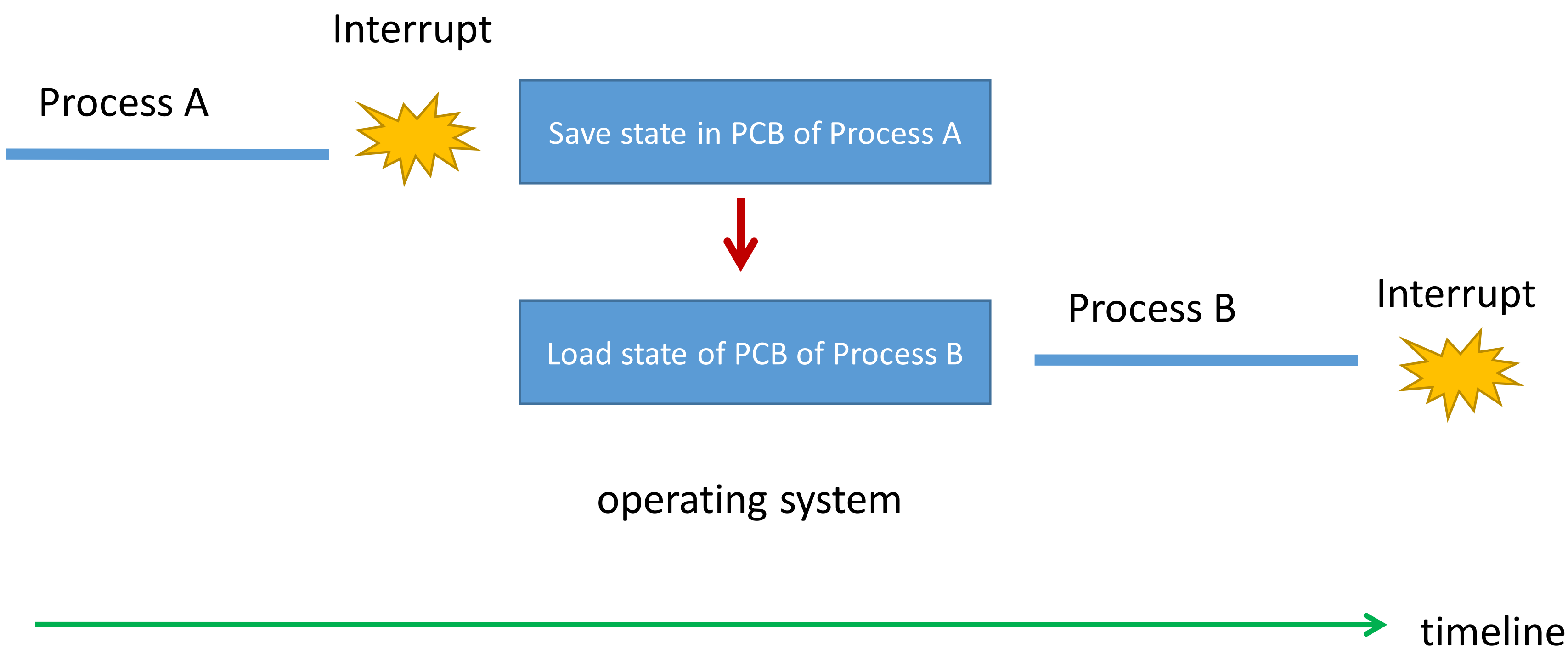
- PC is 126
- `ADD` has occurred but not `STOR`
- Next scheduled process alters registers
- When our process returns to the CPU…
- this is what it finds

Executing Instructions




```
…
LOAD R1, b
LOAD R2, a
ADD R3, R2, R1
STOR R3, c
…
```

CPU Process Switching (Illustration)



Note: switching a process generates management overhead that prevents execution of useful work.

Context Switch

When process P_i removed from CPU,

- Register values stored in PCB_i
- PC stored in PCB_i
- PCB_i state changed to *ready*
- Kernel schedules next process
  - May be same process is only one is currently executing
  - Still follows same rules

Context Switch

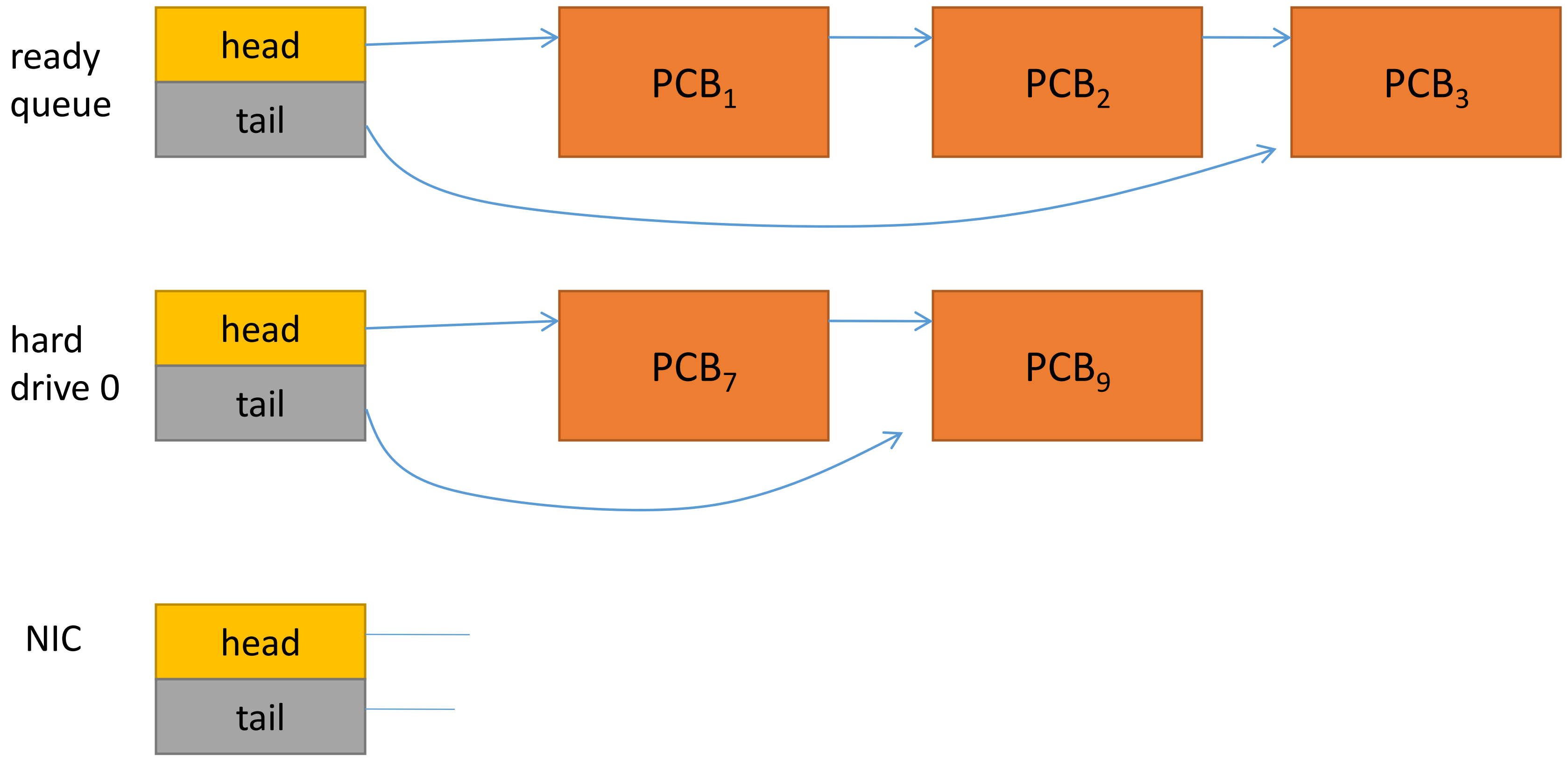
When process P_j returned to CPU,

- Register values restored from PCB_j
- PCB_j state changed to *running*
- CPU mode changed to User-mode
- PC copied from PCB_j to CPU
  - Jumps back to process
  - Last thing to happen

Process Scheduling Queues

- OS maintains multiple queues to manage running processes.
  - each lists PCBs of processes
  - Job queue
  - Ready queue
  - Device queues
- OS moves processes to different queues depending on their status.

Process Queues in Memory



Process Creation, 1

- During boot, one process is created
  - PID is 1
  - Usually named “init”
- All other processes are *descendents* of init
- Processes form hierarchy:
  - parents & children
  - all children have one parent
  - parents can have any number of children

Process Creation, 1 (cont.)

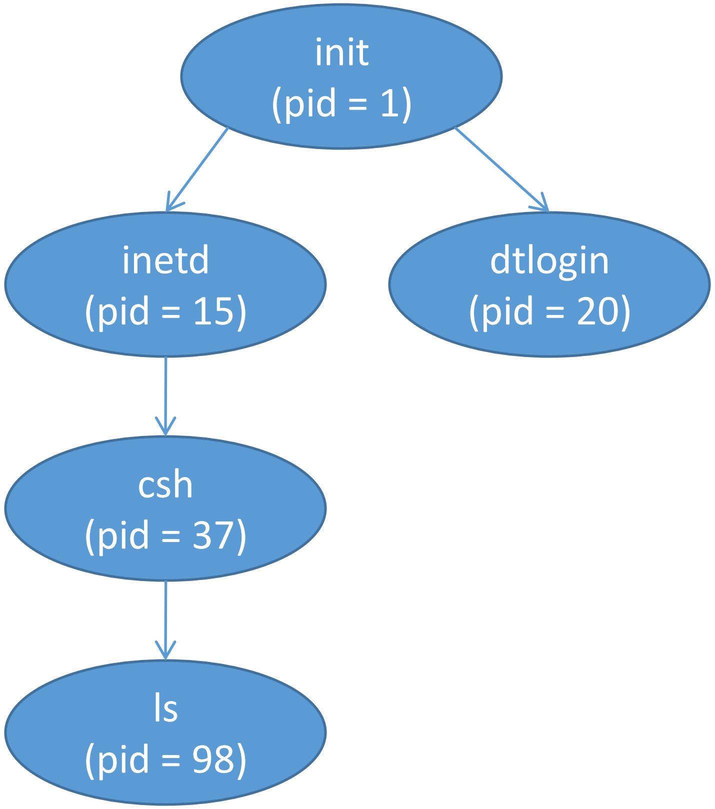
- Address space:
  - parent and child process may share address space
  - create a new address space for child

Process Creation, 2

- System call to create new process:
  `fork()`

  - takes no parameters
  - creates copy of current proc
  - new PCB, new memory, same content
  - memory copy: old to new
  - PCB copy: old to new
  - new PID

Process Tree in UNIX



Process P_{296}

Stack	i	**?**
	j	**?**
Data	g	**?**
Code	LOAD R2, 100	
	STOR R2, j	
	LOAD R3, 5	
	STOR R3, g	
	CALL fork	
	STOR RV, i	
PCB	PID	**296**
	R1	**?**
	R2	**?**
	R3	**?**
	PC	**0**

```
int g;
int main() {
  int i, j;
  j = 100;
  g = 5;
  i = fork();
  printf("%d:%d:%d\n",g,i,j);
  return 0;
}
```

Process P_{296}

Stack	i	**?**
	j	**?**
Data	g	**?**
Code	LOAD R2, 100	
	STOR R2, j	
	LOAD R3, 5	
	STOR R3, g	
	CALL fork	
	STOR RV, i	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**?**
	PC	**1**

```
int g;
int main() {
  int i, j;
  j = 100;
  g = 5;
  i = fork();
  printf("%d:%d:%d\n",g,i,j);
  return 0;
}
```

Process P_{296}

Stack	i	**?**
	j	**100**
Data	g	**?**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**?**
	PC	**2**

```
int g;
int main() {
  int i, j;
  j = 100;
  g = 5;
  i = fork();
  printf("%d:%d:%d\n",g,i,j);
  return 0;
}
```

Process P_{296}

Stack	i	**?**
	j	**100**
Data	g	**?**
Code	LOAD R2, 100	
	STOR R2, j	
	LOAD R3, 5	
	STOR R3, g	
	CALL fork	
	STOR RV, i	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**3**

```
int g;
int main() {
  int i, j;
  j = 100;
  g = 5;
  i = fork();
  printf("%d:%d:%d\n",g,i,j);
  return 0;
}
```

Process P_{296}

Stack	i	**?**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**4**

```
int g;
int main() {
  int i, j;
  j = 100;
  g = 5;
  i = fork();
  printf("%d:%d:%d\n",g,i,j);
  return 0;
}
```

Process P_{296}

Stack	i	**?**
	j	**100**
Data	g	**5**
Code	LOAD R2, 100	
	STOR R2, j	
	LOAD R3, 5	
	STOR R3, g	
	CALL fork	
	STOR RV, i	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**5**

```
int g;
int main() {
   int i, j;
   j = 100;
   g = 5;
   i = fork();
   printf("%d:%d:%d\n",g,i,j);
   return 0;
}
```

Process P_{296} and Process P_{321}

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**296**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

← parent

child →

1 call, 2 returns

Independent processes

Execute in any order

Can be interleaved

Assume child executes 1^{st}

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**321**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

The `fork()`

- Local variables survive
- Global variables survive
- File descriptor table survives
- PCB
  - Registers survive
  - Stack pointer doesn’t survive
  - PC doesn’t survive
  - PID doesn’t survive

Process P_{296} and Process P_{321}

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**296**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

Child didn't call fork()

Child gets a zero return value

0 => child

Quantum Expires

Parent's turn

Stack	`i`	**0**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**321**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**6**

Process P_{296} and Process P_{321}

Stack	i	**321**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**6**

Parent did call fork()

Parent gets PID of child

!0 => parent

Quantum Expires

Child’s turn

Stack	i	**0**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**321**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**6**

Process P_{296} and Process P_{321}

Process P_{296}

Stack	i	**321**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**296**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**?**

<u>OUTPUT</u>

```
5:0:100

5:321:100
```

<u>OR</u>

```
5:321:100

5:0:100
```

Process P_{321}

Stack	i	**0**
	j	**100**
Data	g	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	PID	**321**
	R1	**?**
	R2	**100**
	R3	**5**
	PC	**?**

Notice:

- Program has 1 `printf()` call
- Output has 2 lines
- 1 process (parent) before `fork()`
- 2 processes (parent, child) after `fork()`
- Parent gets PID of child returned from `fork()`
- Child gets 0 returned from `fork()`
- 1 call to `fork()`, 2 returns
- Now we can clone process

Another Example

```
int main(int argc, char ** argv)
{
  pid_t pid;
  printf("Output line 1\n");
  pid = fork();
  printf("Output line 2 with pid = %d\n", pid);
  return 0;
}
```

What is the output?

Assume parent is PID 94 and child is PID 223.

Another Example

```
int main(int argc, char ** argv)
{
  pid_t pid;
  printf("Output line 1\n");
  pid = fork();
  printf("Output line 2 with pid = %d\n", pid);
  return 0;
}
```

What is the output?

Assume parent is PID 94 and child is PID 223.

```
Output line 1
Output line 2 with pid = 223
Output line 2 with pid = 0
```

Parent printed first

Another Example

```
int main(int argc, char ** argv)
{
  pid_t pid;
  printf("Output line 1\n");
  pid = fork();
  printf("Output line 2 with pid = %d\n", pid);
  return 0;
}
```

What is the output?

Assume parent is PID 94 and child is PID 223.

```
Output line 1
Output line 2 with pid = 0
Output line 2 with pid = 223
```

Child printed first

Normal Usage

```
int main(int argc, char ** argv)
{
  pid_t pid;
  doCommonEntryCode();
  pid = fork();
  if (pid) {
    doParentCode();
  } else {
    doChildCode();
  }
  doDuplicatedExitCode();
  return 0;
}
```

What now?

- We can clone process
- We can create clone of `myshell`
- We want to create `ls` instead
- How?
- Try the `exec()` family of system calls

exec family of functions

int **execl**(const char *path, const char *arg, ...);

int **execlp**(const char *file, const char *arg, ...);

int **execle**(const char *path, const char *arg , ..., char * const envp[]);

int **execv**(const char *path, char *const argv[]);

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

Description

The `exec` family of functions replaces the current process image with a new process image.

The initial argument for these functions is the pathname of a file which is to be executed.

Description

The `const char *arg` and subsequent ellipses in the `execl()`, `execlp()`, and `execle()` functions can be thought of as `arg0`, `arg1`, ..., `argn`. They describe a list of one or more pointers to null-terminated strings that represent the argument list available to the executed program. The first argument should point to the file name associated with the file being executed. The list of arguments must be terminated by a `NULL` pointer, and, since these are variadic functions, this pointer must be cast `(char *) NULL`.

printargs.c

```
#include <stdio.h>
int main(int argc, char ** argv) {
  int i;
  for (i = 0; i < argc; i++)
    printf("argv[%d] = |%s|\n",
           i, argv[i]);
  return 0;
}
```

`gcc -g -Wall printargs.c -o pa`

Executing `pa`

```
> pa
argv[0] = |pa|
> pa -m file
argv[0] = |pa|
argv[1] = |-m|
argv[2] = |file|
>
```

More Forking

```
pid = fork();
if (pid) {
  printf("parent: pid = %d\n", pid);
} else {
  execl("./pa", "pa", NULL);
  printf("child: pid = %d\n", pid);
 exit(1);
}
printf("more output\n");
```

Process P_{296} and Process P_{321}

after fork

2 copies of parent

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**296**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**321**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

Process P_{296} and Process P_{321}

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**296**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

after execl()

1 parent executing original code

1 child executing pa

Stack		
Data		
Code	`exec for pa`	
	…	
	…	
	…	
	…	
	…	
PCB	`PID`	**321**
	`R1`	**?**
	`R2`	**?**
	`R3`	**?**
	`PC`	**0**

Process P_{296} and Process P_{321}

Stack	`i`	**?**
	`j`	**100**
Data	`g`	**5**
Code	`LOAD R2, 100`	
	`STOR R2, j`	
	`LOAD R3, 5`	
	`STOR R3, g`	
	`CALL fork`	
	`STOR RV, i`	
PCB	`PID`	**296**
	`R1`	**?**
	`R2`	**100**
	`R3`	**5**
	`PC`	**5**

after execl()

Stack, data, initialized

Code for pa overwrites code for parent

PCB initialized

Stack		
Data		
Code	`exec for pa`	
	…	
	…	
	…	
	…	
	…	
PCB	`PID`	**321**
	`R1`	**?**
	`R2`	**?**
	`R3`	**?**
	`PC`	**0**

Output

```
parent: pid = 1093
argv[0] = |pa|
more output
```

OR

```
parent: pid = 1093
more output
argv[0] = |pa|
```

output determined by quantum expiration times

More Forking

```
pid = fork();
if (pid) {
  printf("parent: pid = %d\n", pid);
} else {
  execl("./pa", "pa", "-m", "file",
        NULL);
  printf("child: pid = %d\n", pid);
 exit(1);
}
printf("more output\n");
```

Sample Output

```
parent pid = 1093
argv[0] = |pa|
more output
argv[1] = |-m|
argv[2] = |file|
```

```
output still
determined by
quantum expiration
times
```

Why so many `exec()`s?

Allow argument passing in different forms

- `execl()` – give path to exe, separate args
- `execlp()` – give name of exe, separate args
- `execv()` – give path to exe, arg vector
- `execvp()` – give name of exe, arg vector

Project 1: Check the values produced from the first part of the project. What form are they in? Is that useful?

What happens after `exec()`?

- Local variables don’t survive
- Global variables don’t survive
- File descriptor table survives
- PCB
  - Registers don’t survive
  - Stack pointer doesn’t survive
  - PC doesn’t survive
  - PID survives

Summary

- A process is a running program in memory.
- The OS creates a PCB for each process.
- The process goes through different stages during its life cycle.
  - may be performing IO, then it is blocked
  - may be running or waiting or terminating or starting
- A process is created through fork().
  - except for the first process, called init
- System call exec loads the code of a process into memory.