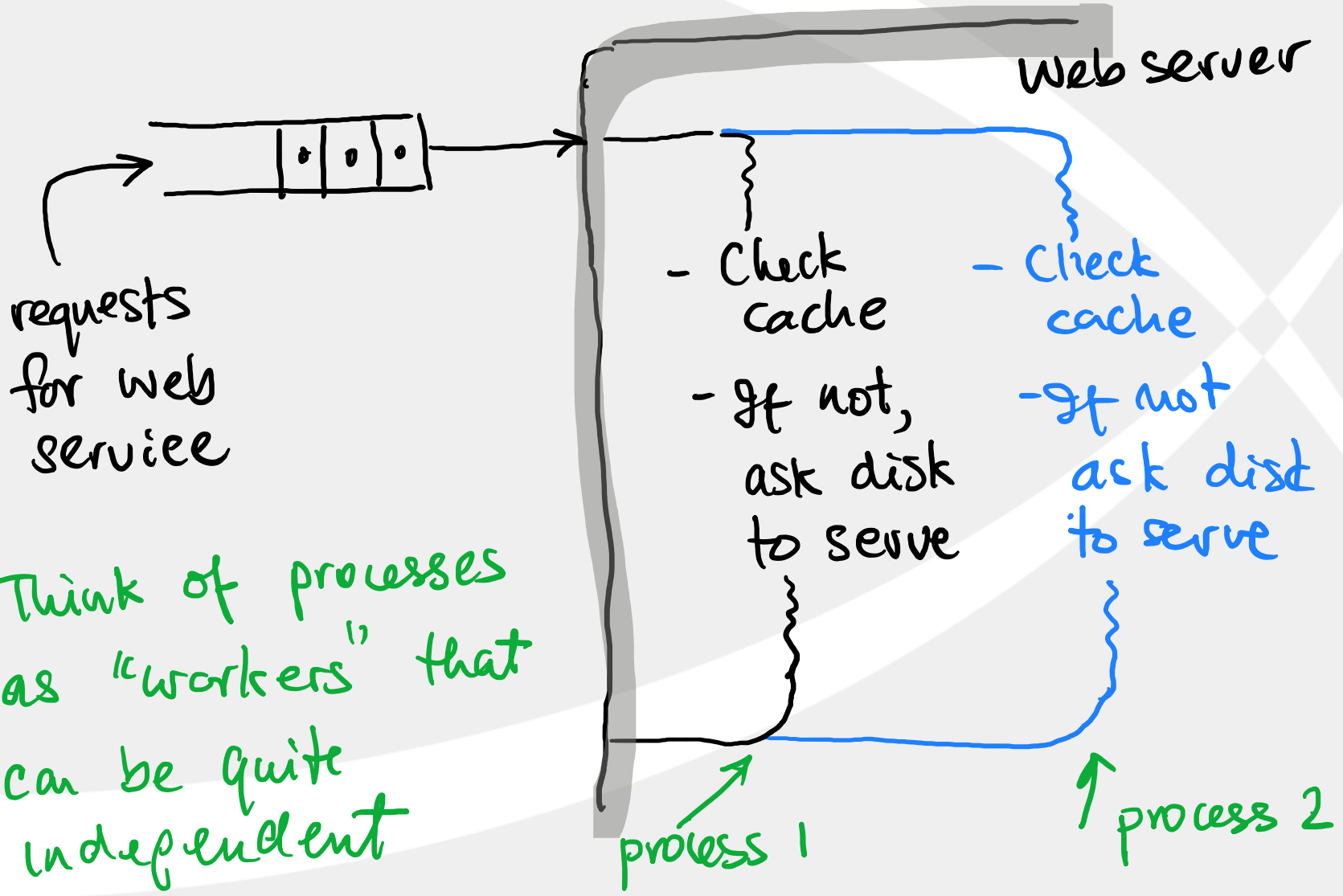
Processes and Threads

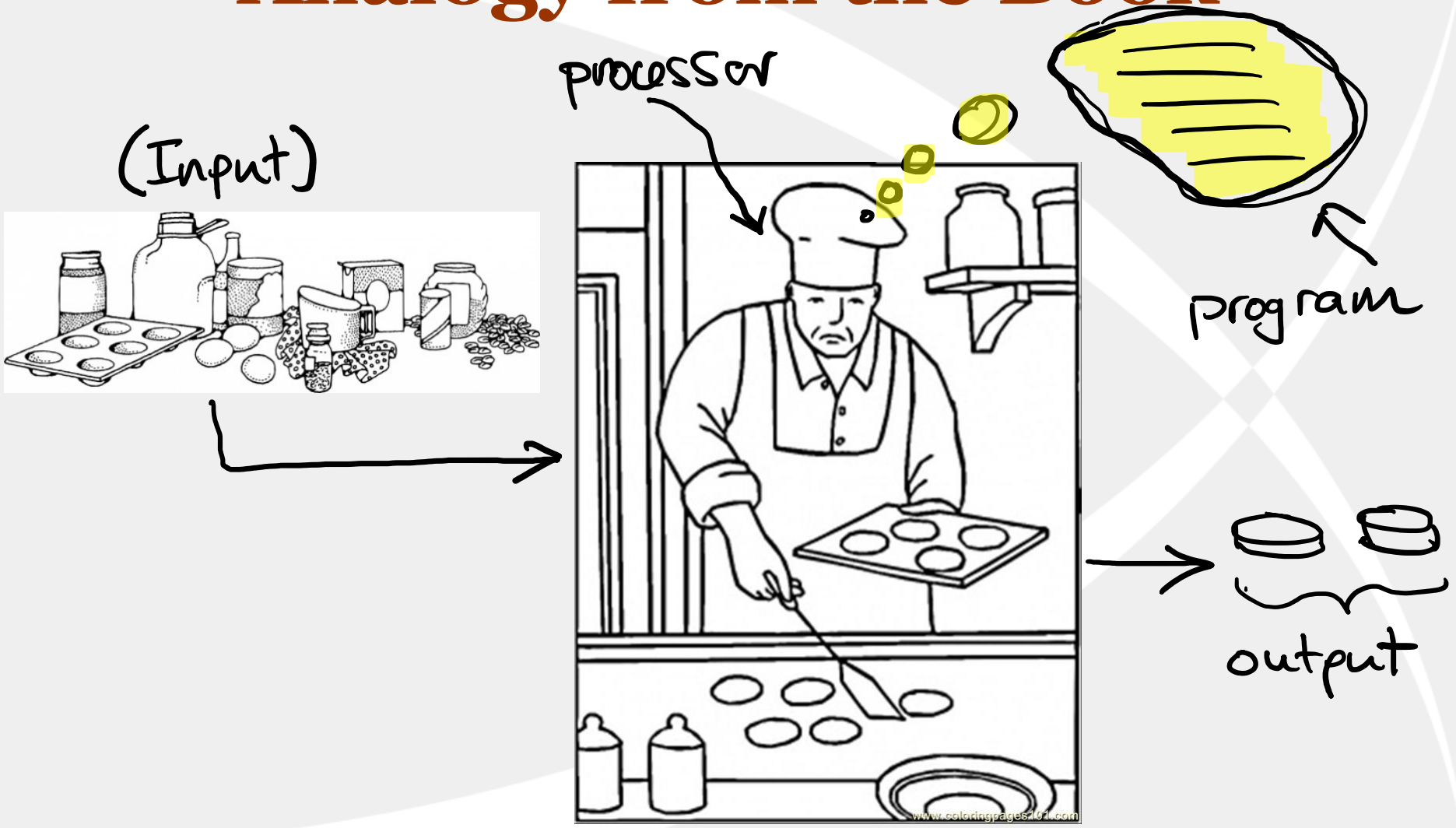
What is a *process*?

- Process
  - An abstraction of a running program
  - Modern OS hinges on this concept
  - Supports the ability to have (pseudo) concurrent operation
- To illustrate, lets consider a web server

Scenario: Web Server

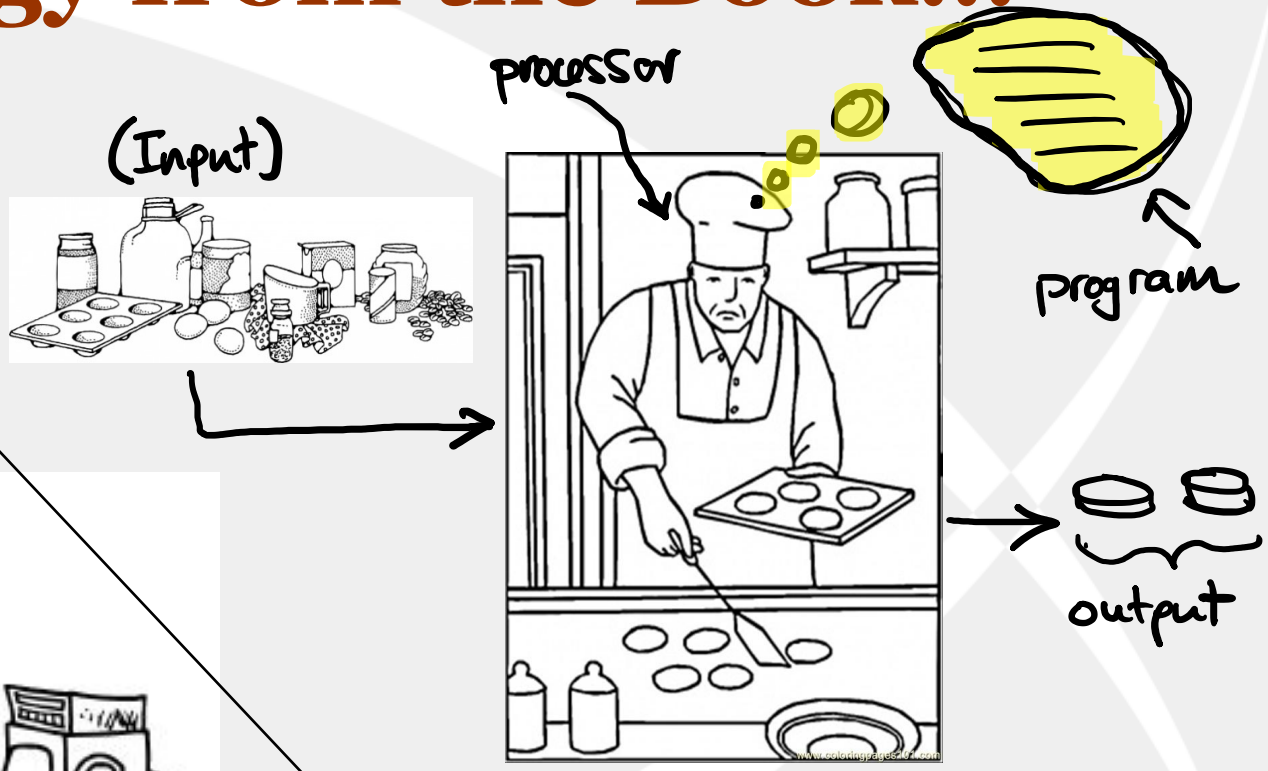


Analogy from the Book

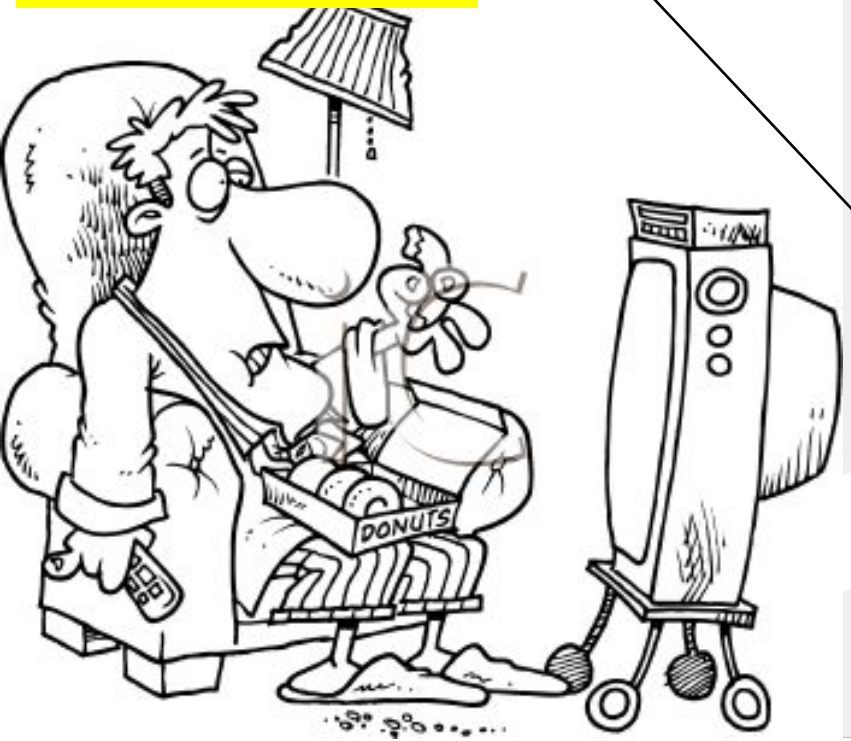


"Process" is the activity of baking the cookies

Analogy from the Book...



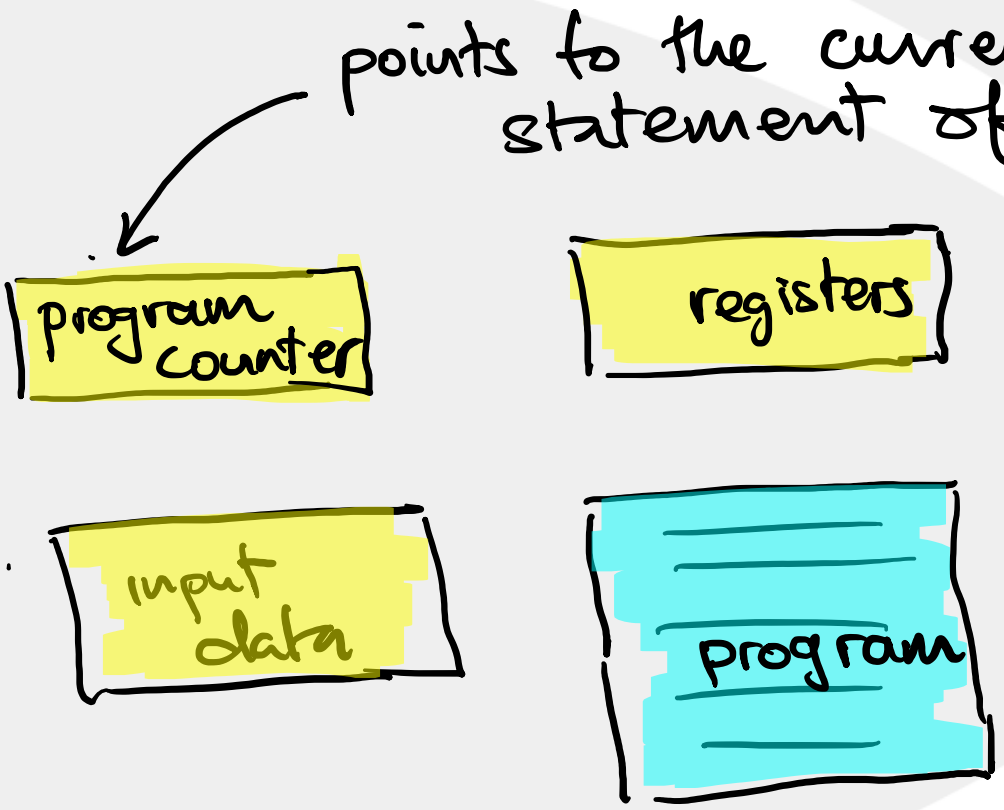
"Process" is the activity of watching football



"Process" is the activity of baking the cookies

Single person needs to switch between the two different processes..

Key Elements of a Process



some key elements missing... what are they?

Multiprogramming

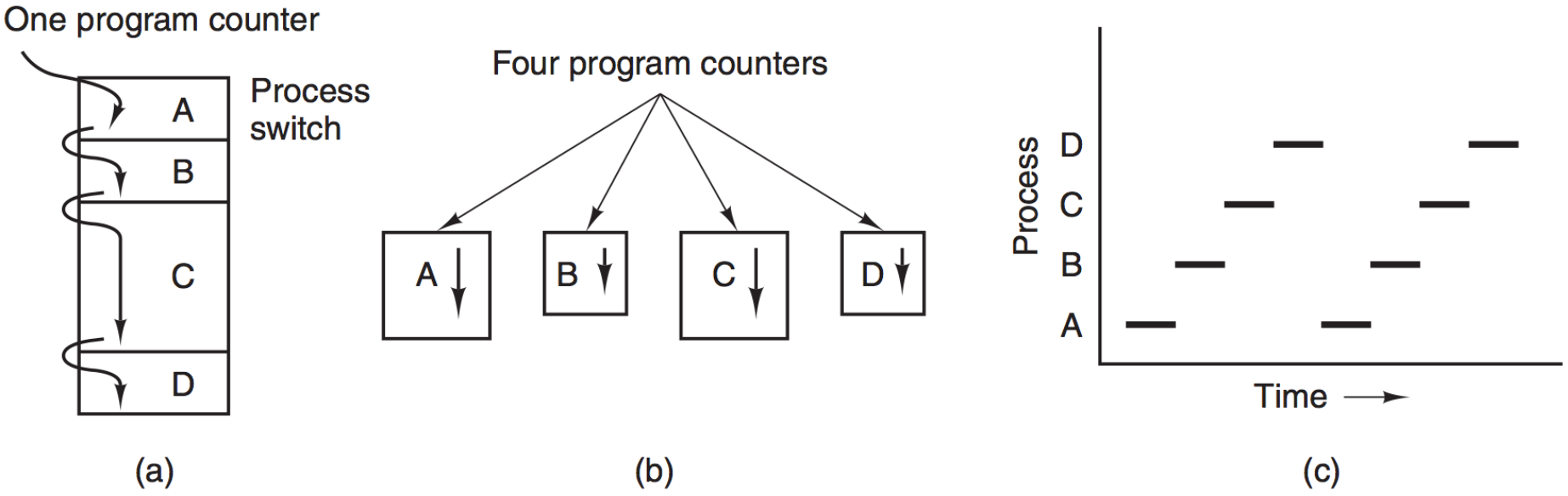


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Write the pseudo code for the “program” switching; what do we need to save and restore?

Uniprogramming vs. Multiprogramming

Runs a single program at a time

multiprograming

Program X

Program Y

Program Z

No need for the process concept

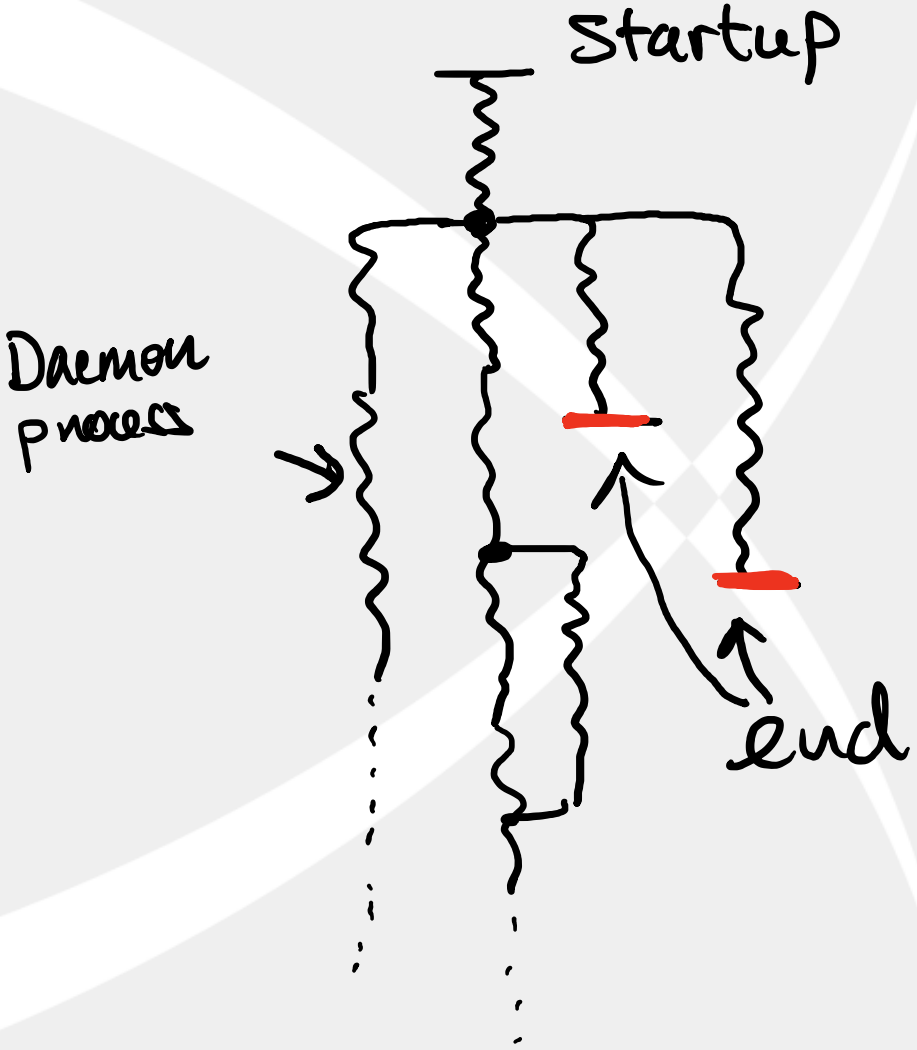
switch between different programs that are elisible to run.

Process management issues?

- Process management issues:
  - Lifecycle management
  - Precedence management (flow management)
- Lifecycle management:
  - Process creation
  - Process state changes, reasons (what happens in the middle!)
  - Process termination

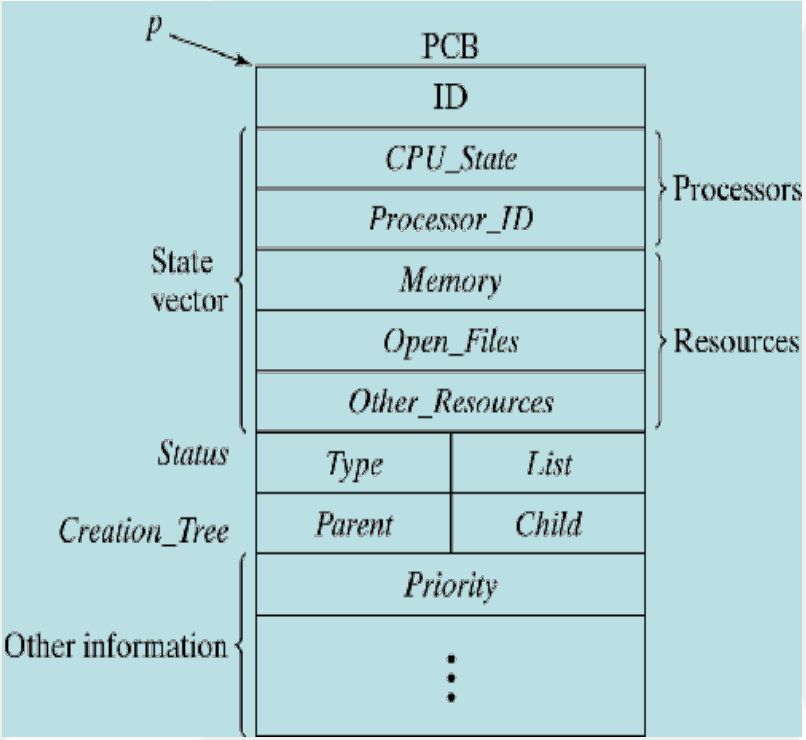
Process Creation

- Four principle events creating processes
  - System initialization
  - Execution of a process creation system-call
  - User request to create a new process
  - Initiation of a batch job



How is a process represented?

- Information: state & control
- Process Control Block (PCB)
  - Identifier
  - State Vector = Information necessary to run process p
  - Status
  - Creation tree
  - Priority
  - Other information



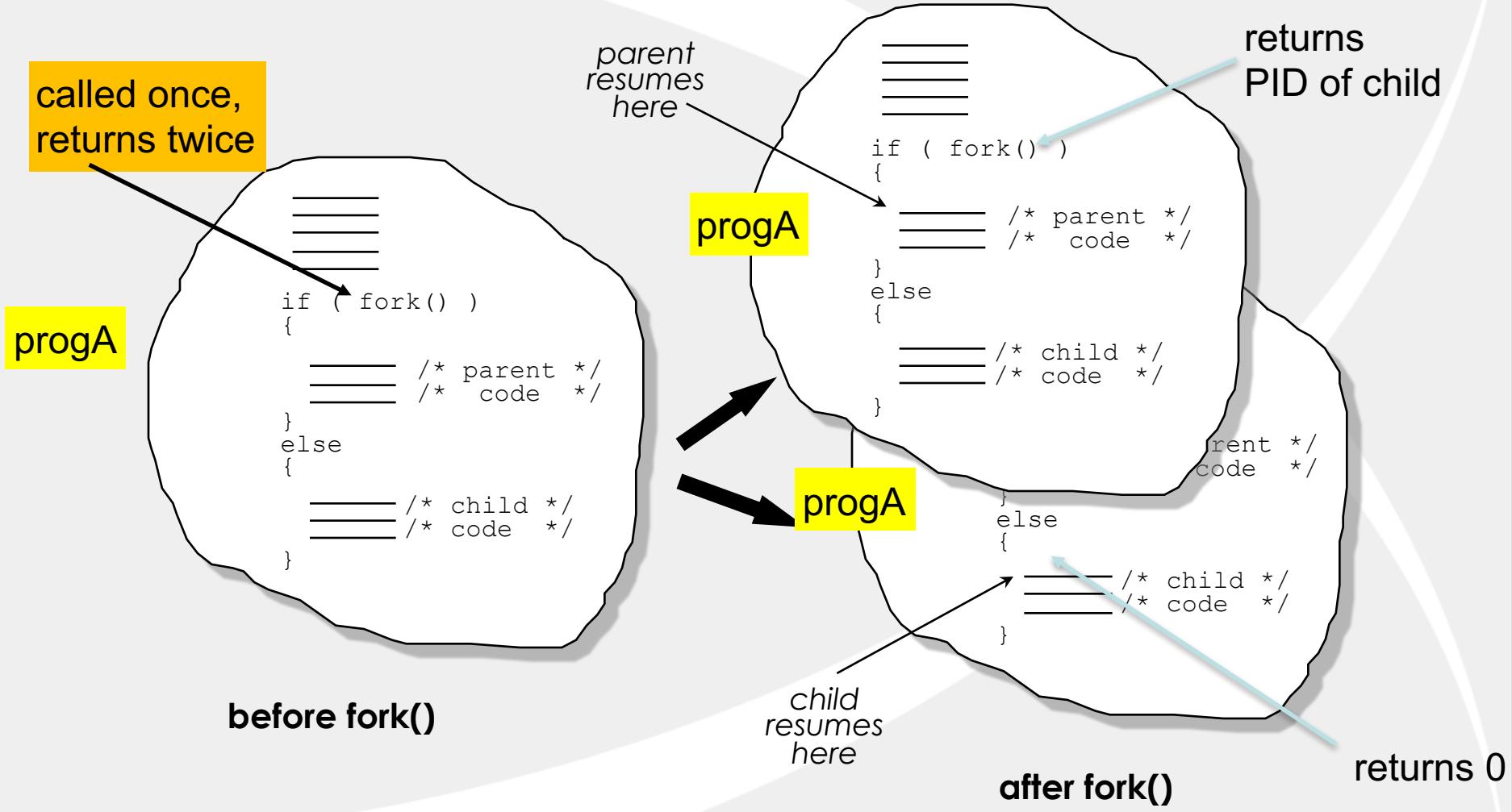
Lifecycle: Create process

- Two ways of creating a new process:
  - *Build one from scratch*: ← Windows approach
    - load *code* and *data* into memory
    - create (empty) a *dynamic memory workspace (heap)*
    - create and initialize the *process control block*
    - make process known to process scheduler (dispatcher)
  - *Clone an existing one*: ← UNIX approach
    - stop current process and save its state
    - make a copy of *code*, *data*, *heap* and *process control block*
    - make process known to process scheduler (dispatcher)

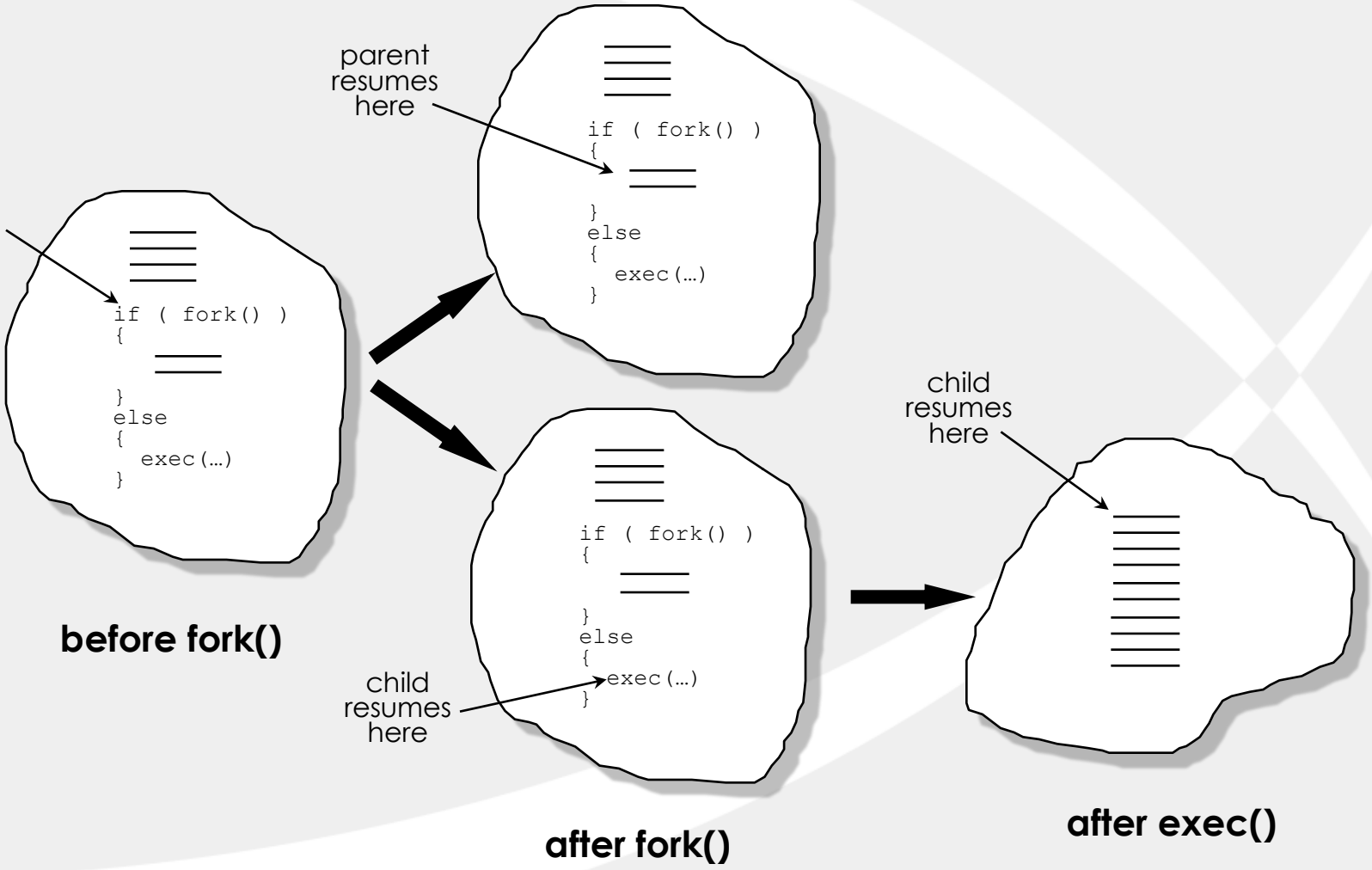
UNIX process creation

- In UNIX, the **`fork()`** system call is used to create processes
  - **`fork()`** creates an identical copy of the calling process
  - after the **`fork()`**, the *parent* continues running concurrently with its *child* competing equally for the CPU

UNIX process creation...



A typical use of `fork()`



Example

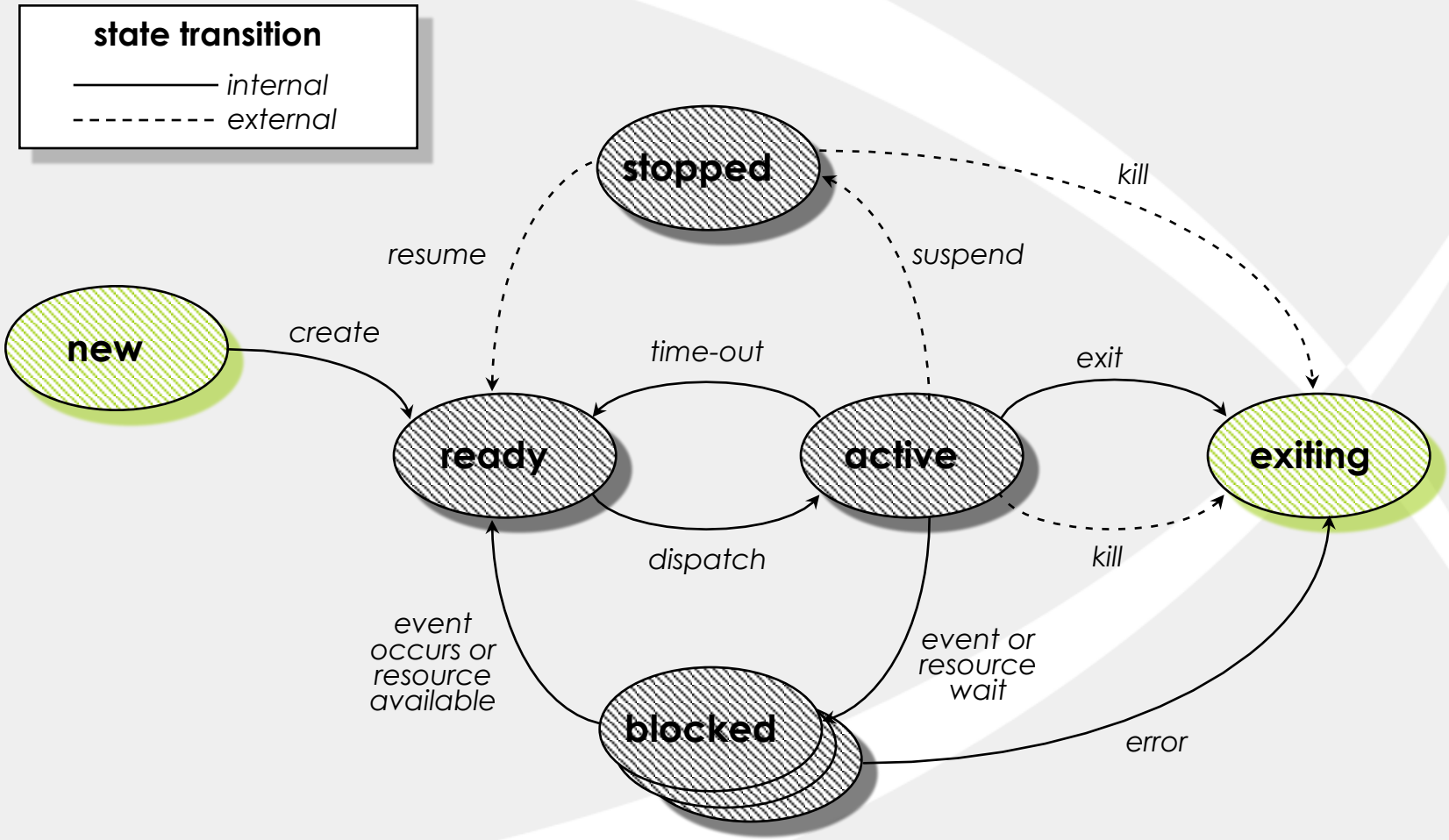
- What is the output of the following simple C program?

```
main() {
    int i;
    i = 10;
    if (fork() == 0) i += 20;
    printf(" %d ", i);
}
```

Lifecycle: After creation

- After creation, process can experience various conditions:
  - No resources to run (e.g., no processor, memory)
  - Waiting for a resource or event
  - Completed the task and exit
  - Temporarily suspend waiting for a condition
- → Process should be in different states

Process state diagram



Process states

- A process can be in many different states:
  - **New**—a process being created but not yet included in the pool of executable processes (*resource acquisition*)
  - **Ready**—processes are prepared to execute when given the opportunity
  - **Active**—the process that is currently being executed by the CPU
  - **Blocked**—a process that cannot execute until some event occurs
  - **Stopped**—a special case of **blocked** where the process is suspended by the operator or the user
  - **Exiting**—a process that is about to be removed from the pool of executable processes (*resource release*)

Example

- Following are code segments from a process that is already created and eligible to run or running

```
    ….
(a) i = i + j * 10;
    a[i] = b[j] * c[i];
(b) read(scale);        // reading standard
    input
                        for a variable
    ….
(c) wait (mutex);       // waiting on a mutual
                        exclusion variable
```

- What are the possible process states at (a), (b), and (c)?

Lifecycle: Process termination

- A process enters the *exiting* state for one of the following reasons
  - normal completion: A process executes a system call for termination (e.g., in UNIX **`exit()`** is called).
  - abnormal termination:
    - programming errors
      - *run time*
      - *I/O*
    - user intervention

Tiny Shell

- Repeat
  - Show a prompt
  - Read a line of input
  - Run the given command in a child process

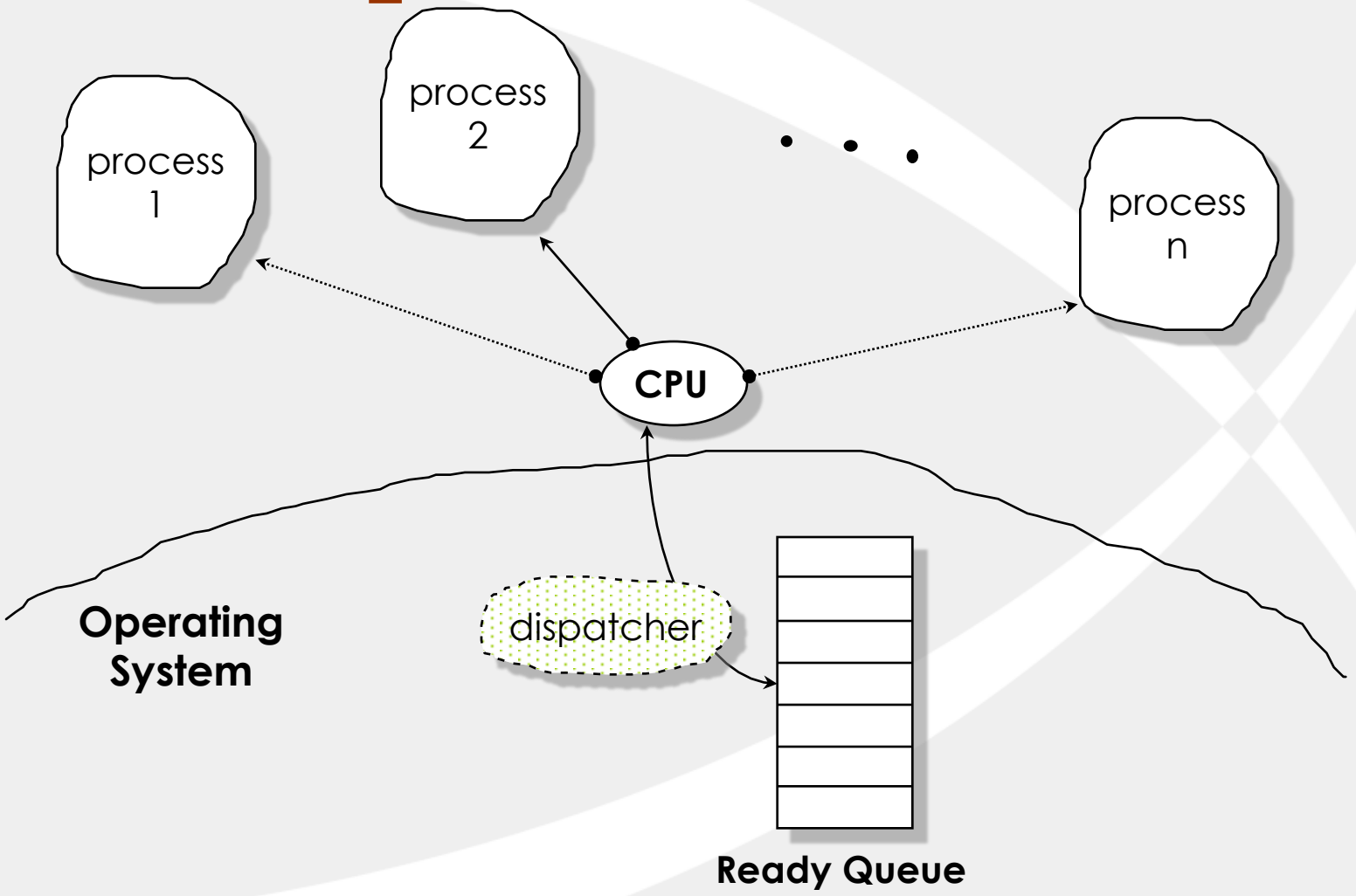
```
while (1) {
    printf("Shell > ")
    getline(line)
    if (strlen(line) > 1) {
        if (fork() == 0) {
            exec(line)
        }
        wait(child)
    }
}
```

Implementing processes

- With multi-programming, we have several processes concurrently executing
- OS is responsible:
  - Dynamically selecting the next process to run
  - Rescheduling performed by the dispatcher
- Dispatcher given by:

```
loop forever {
        run the process for a while.
        stop process and save its state.
        load state of another process.
}
```

Dispatcher at work



Dispatcher: Controlling the CPU?

- CPU can only do one thing at a time
- While user process running, dispatcher (OS) is NOT running
- How does the dispatcher regain control?
  - Trust the process to wake up the dispatcher when done (*sleeping beauty approach*).
  - Provide a mechanism to wake up the dispatcher (*alarm clock*).
- Obviously, the *alarm clock* approach is better. Why?

How is an alarm event handled?

- Context switch happens:
  - OS saves the state of the *active* process and restores the state of the *interrupt service routine*
  - Simultaneously, CPU switches to *supervisory* mode
- What must get saved? *Everything that the next process could or will damage*. For example:
    - *Program counter (PC)*
    - *Program status word (PSW)*
    - *CPU registers (general purpose, floating-point)*
    - *File access pointer(s)*
    - *Memory (perhaps?)*
- While saving the state, the operating system should mask (disable) *all* interrupts.

Memory: *to save* or *NOT to save*

- Here are the possibilities:
  - Save *all* memory onto disk.

    Could be *very* time-consuming. E.g., assume data transfers to disk at 1MB/sec. How long does saving a 4MB process take?

  - Don't save memory; trust next process.

    This is the approach taken by (older) PC and Mac OSes.

  - Isolate (protect) memory from next process.

    This is *memory management*, to be covered later

CPU switching among processes