Processes - A System View

Concurrency & Context Switching

Process Control Block

What's in it and why? How is it used? Who sees it?

5 State Process Model

State Labels. Causes of State Transitions. Impossible Transitions.

Zombies and Orphans

How does **fork** work?

- Parent
  `mypid = 4, myppid = 1`

```
int forked_pid , wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```

- Child
  `mypid = 6, myppid = 4`

```
int forked_pid, wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```

How does **fork** really work?

- Parent
  `mypid = 4, myppid = 1`

```
int forked_pid , wait_pid;
int status = 0;

if (forked_pid = fork()) {
    /* parent */
    …..
    wait_pid = wait(&status);
} else {
    /* child */
    …..
    exit(status);
}
```

- Child
  `mypid = 6, myppid = 4`

```
int forked_pid , wait_pid;
int status = 0;
```

What the fork?

- Concurrency
  - What is a sequential program?
    - A single thread of control that executes one instruction
    - When it is finished, it executes the next logical instruction
  - What is a concurrent program?
    - A collection of autonomous sequential programs, executing (logically) in parallel
- What does this gain us?
  - The appearance that multiple actions are occurring at the same time

What is fork good for?

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

int main() {
    pid_t pid;
    int i;


        if(pid = fork()) {          /* parent */
            parentProcedures();
        }
        else {                       /* child */
            childProcedures();
        }

    return 0;
}
```

What is fork good for?

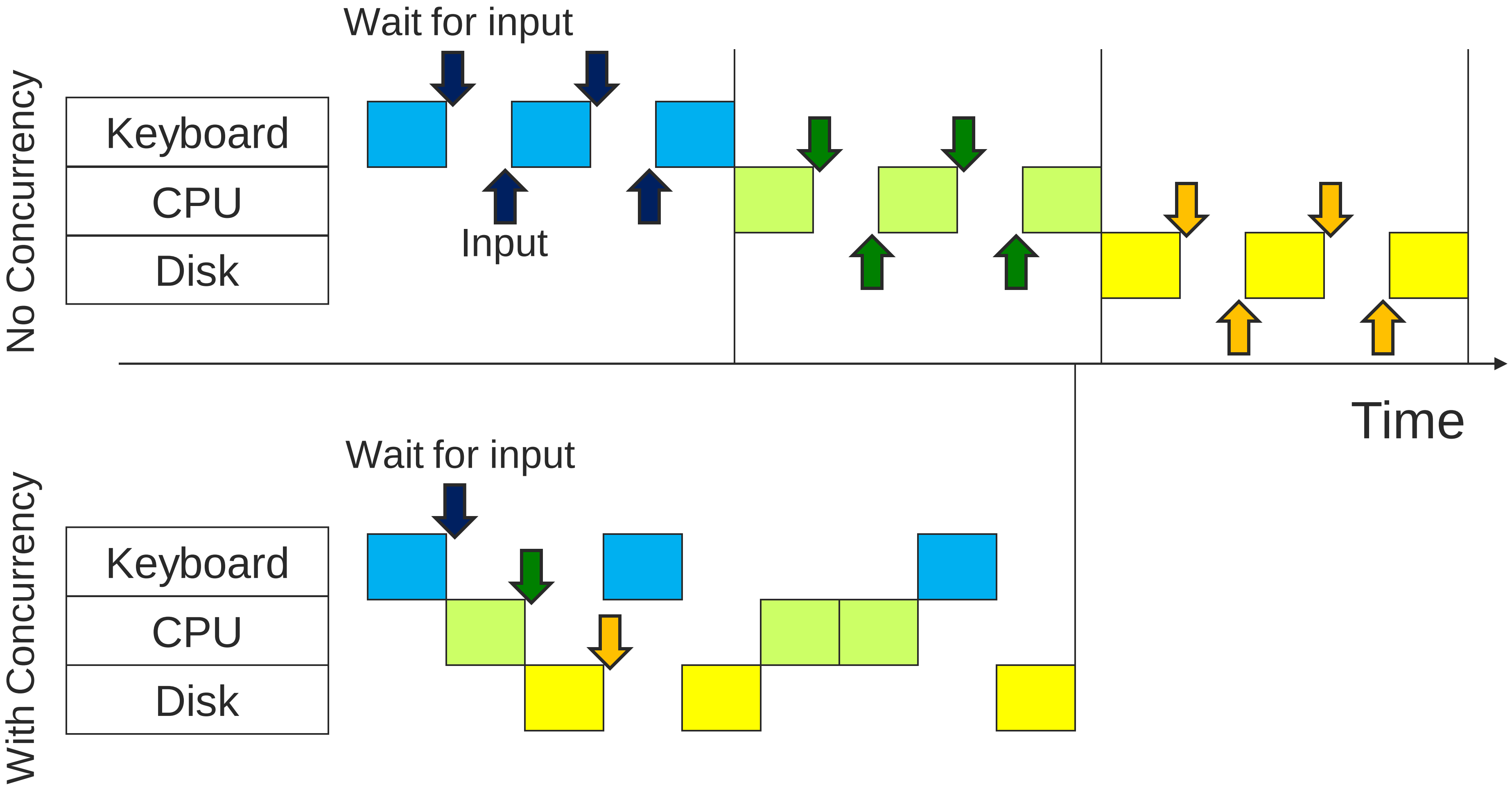
```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>

int main() {
    pid_t pid;
    int i;
    while (1) {
        /* wait for new clients */
        if(pid = fork()) {          /* parent */
            /* reset server */
        }
        else {                      /* child */
            /* handle new client */
        }
    }
    return 0;
}
```

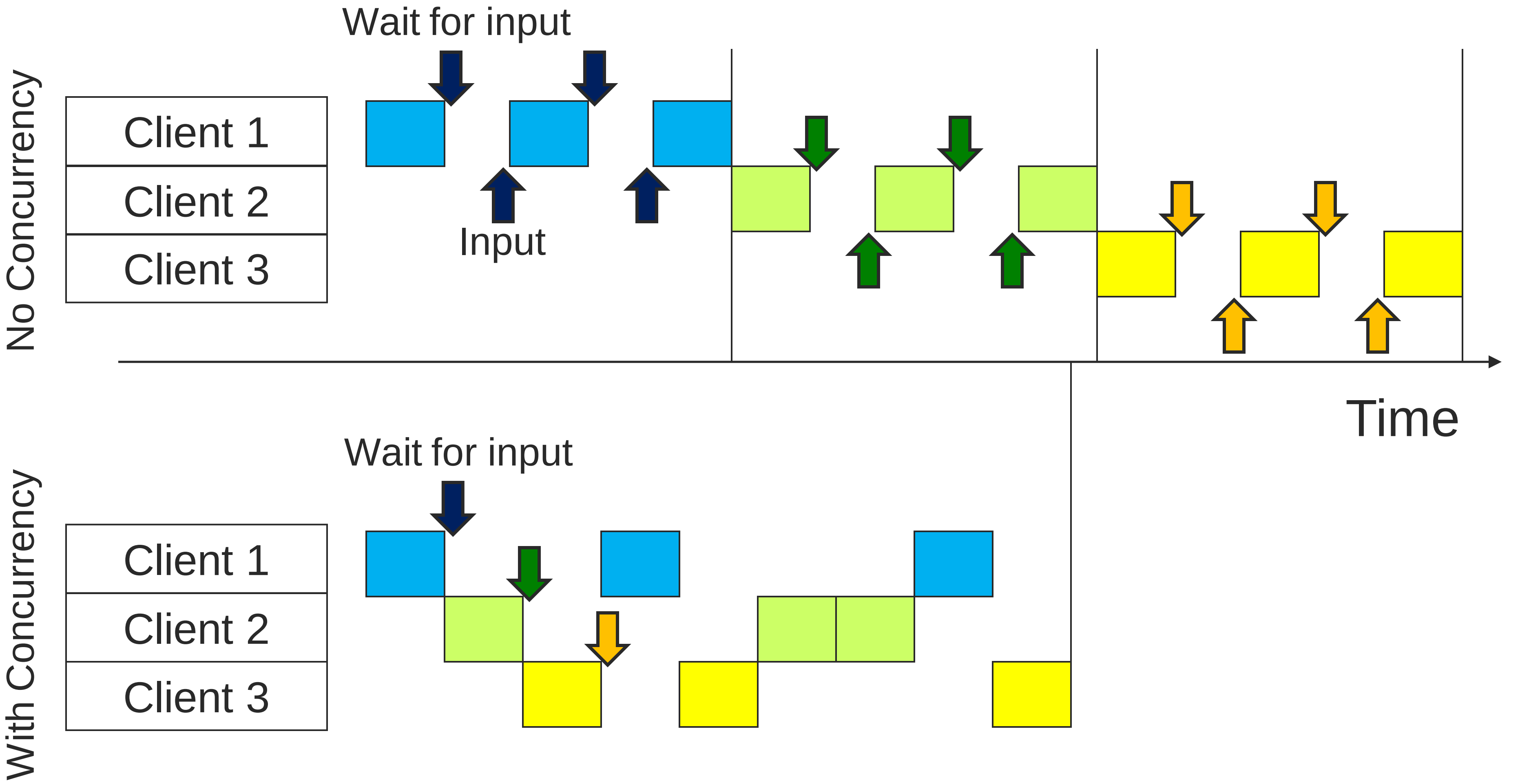
Why Concurrency?

- Natural Application Structure
  - The world is not sequential!
  - Easier to program multiple independent and concurrent activities
- Better resource utilization
  - Resources unused by one application can be used by the others
- Better average response time
  - No need to wait for other applications to complete

Benefits of Concurrency



Benefits of Concurrency



On a single CPU system…

- Only one process can use the CPU at a time
  - Uniprogramming
    - Only one process resident at a time

  … But we want the appearance of every process running at the same time
- How can we manage CPU usage?
  - “Resource Management”

On a single CPU system…

- Your process is currently using the CPU

```
long count = 0;
while(count >=0)
   count ++;
```

- What are other processes doing?

On a single CPU system…

- Answer
  - Nothing
- What can the OS do to help?
  - Naively… Put the current process on 'pause'
- What are our options?

O/S : I need the CPU

1. Time slicing
   - Use a HW timer to generate a HW interrupt
2. Multiprogramming
   - Multiple processes resident at a time
   - Wait until the process issues a system call
     - e.g., I/O request
3. Cooperative Multitasking
   - Let the user process yield the CPU

Time Slicing

- A Process loses the CPU when its time quanta has expired

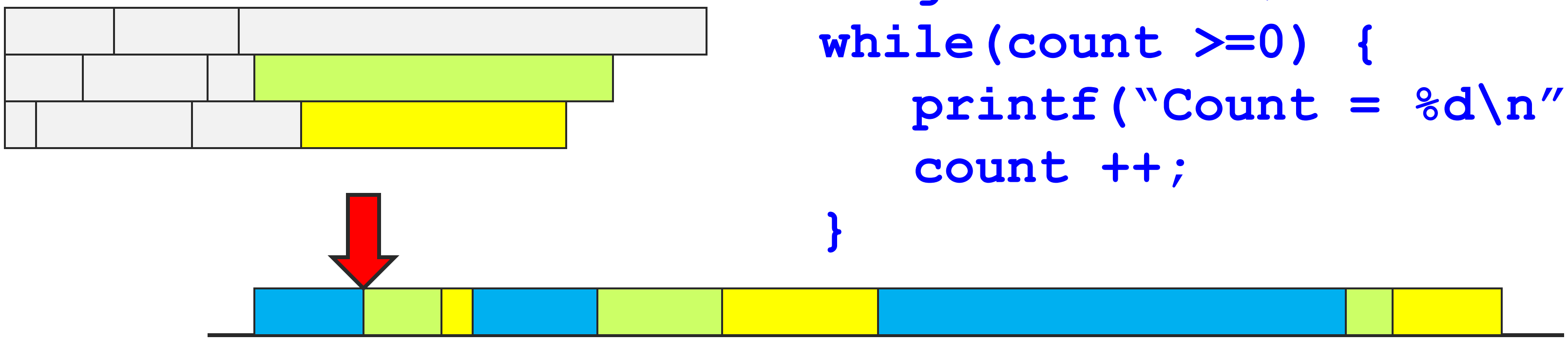
```
long count = 0;
while(count >=0)
   count ++;
```

- Advantages?
- Disadvantages?

Multiprogramming

- Wait until system call

```
long count = 0;
while(count >=0) {
    printf("Count = %d\n", cnt);
    count ++;
}
```



- Advantages?
- Disadvantages?

Cooperative Multitasking

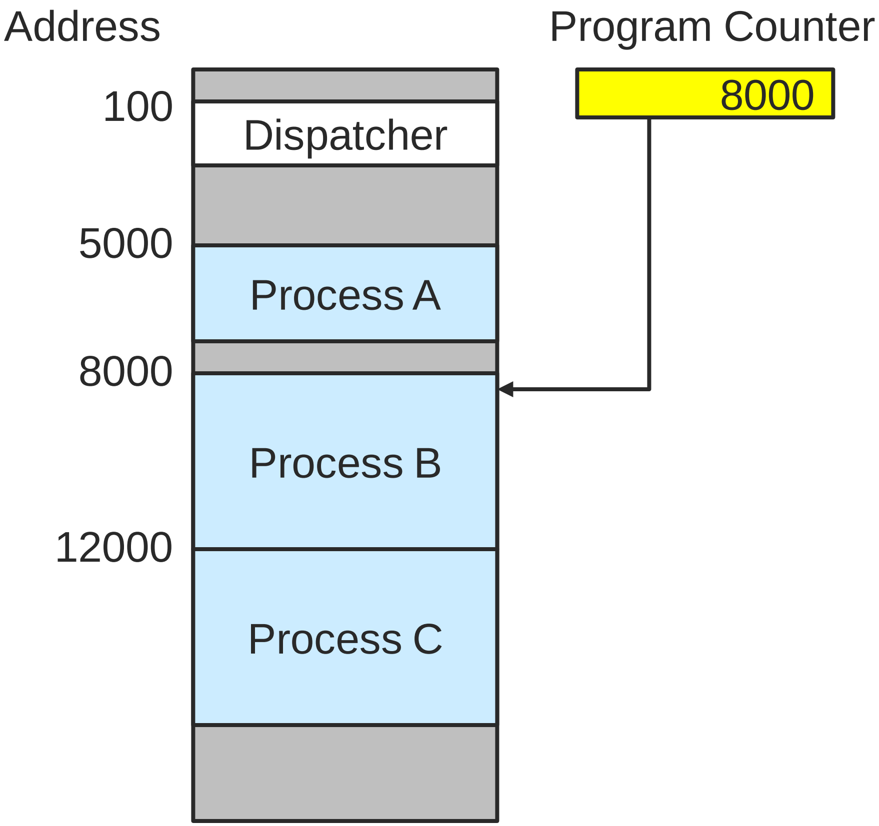
- Wait until the process gives up the CPU

```
long count = 0;
while(count >=0) {
   count ++;
   if(count % 10000 == 0)
      yield();
}
```

- Advantages?
- Disadvantages?

Context Switch: In a simple O/S (no virtual memory)

- Context switch
  - The act of removing one process from the running state and replacing it with another



Context Switch

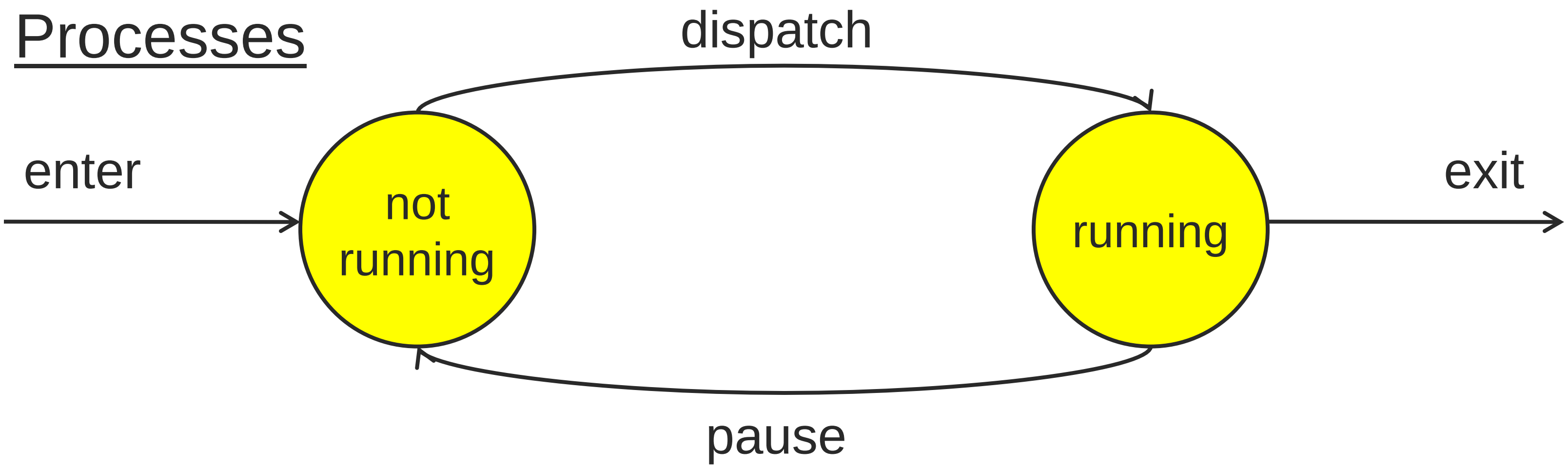
- Overhead to re-assign CPU to another user process

- What activities are required?

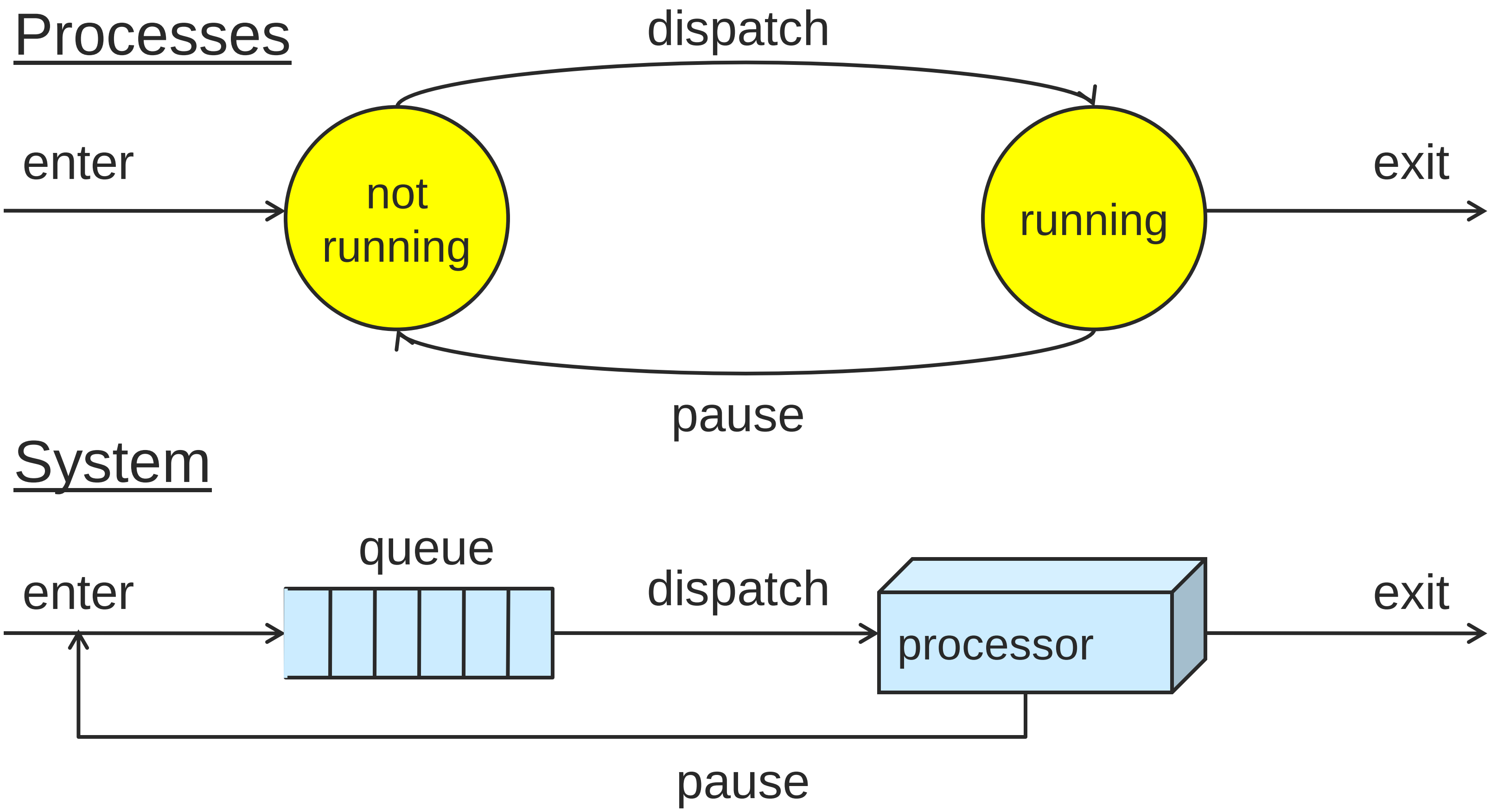
Context Switch

- Overhead to re-assign CPU to another user process
  - Capture state of the user's processes so that we can restart it later (CPU Registers)
  - Queue Management
  - Accounting
  - Scheduler chooses next process
  - Run next process

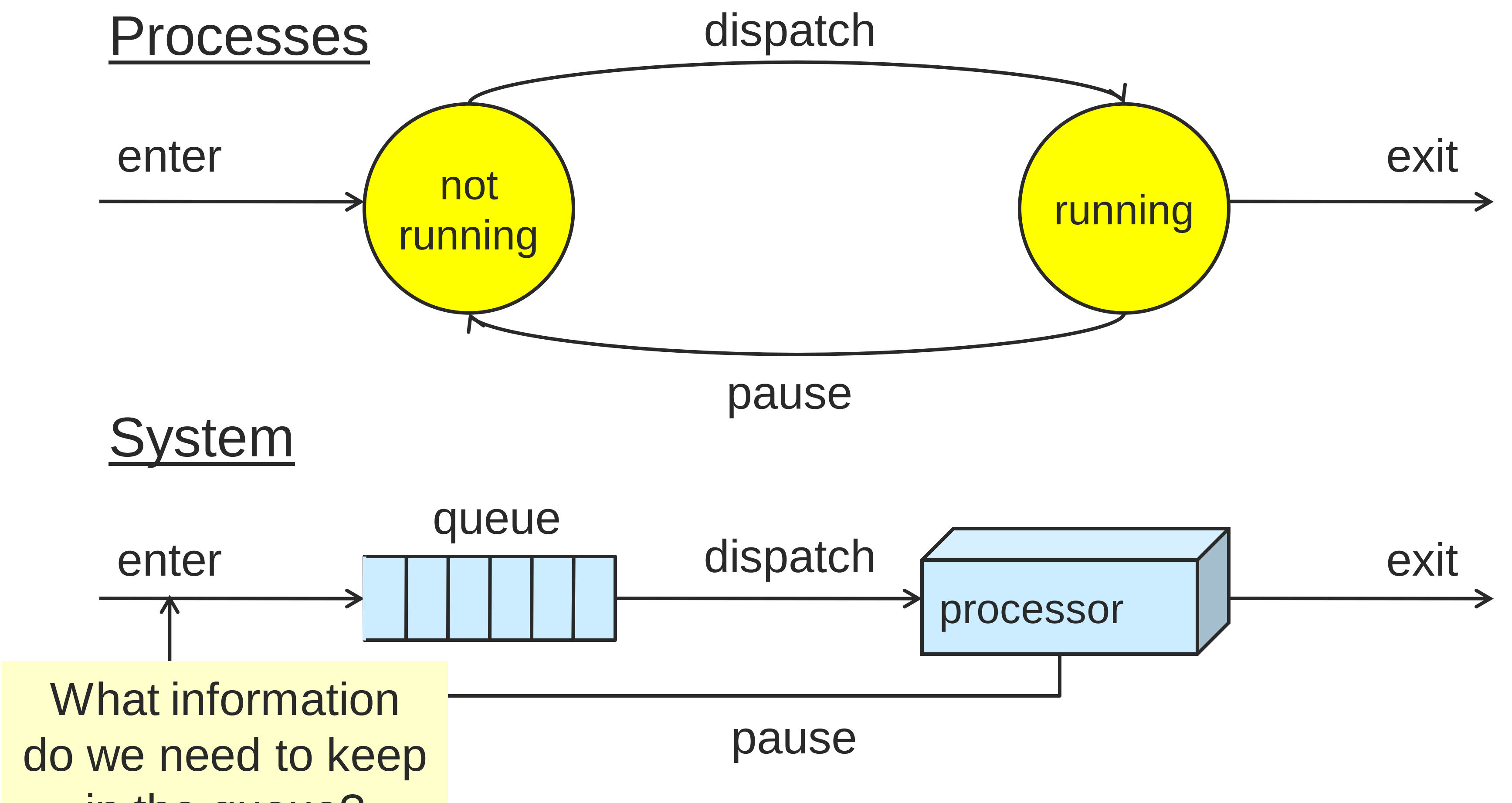
2 State Model



2 State Model



2 State Model



Process Control Block (PCB)

- In-memory system structure
  - User processes cannot access it
  - Identifiers
    - pid & ppid
  - Processor State Information
    - User-visible registers, control and status, stack
  - Scheduling information
    - Process state, priority, …, waiting for event info

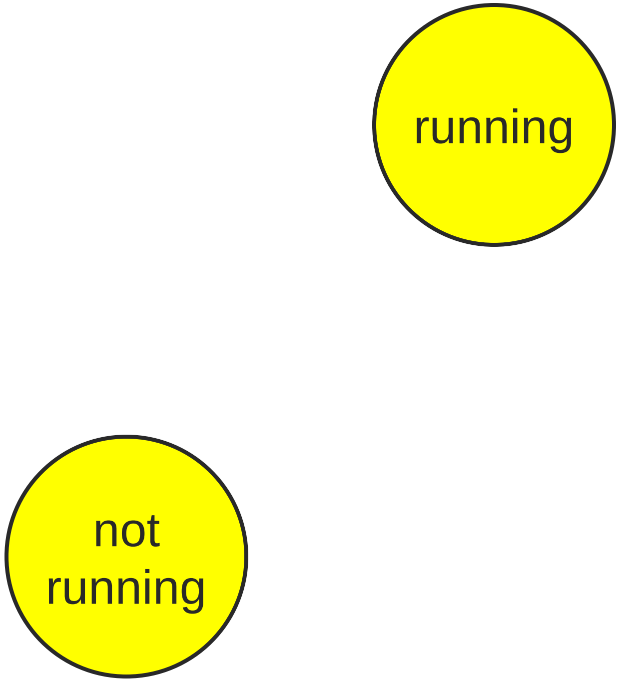
PCB (more)

- Inter-process communication
  - Signals
- Privileges
  - CPU instructions, memory
- Memory Management
  - Segments, VM control 'page tables'
- Resource Ownership and utilization

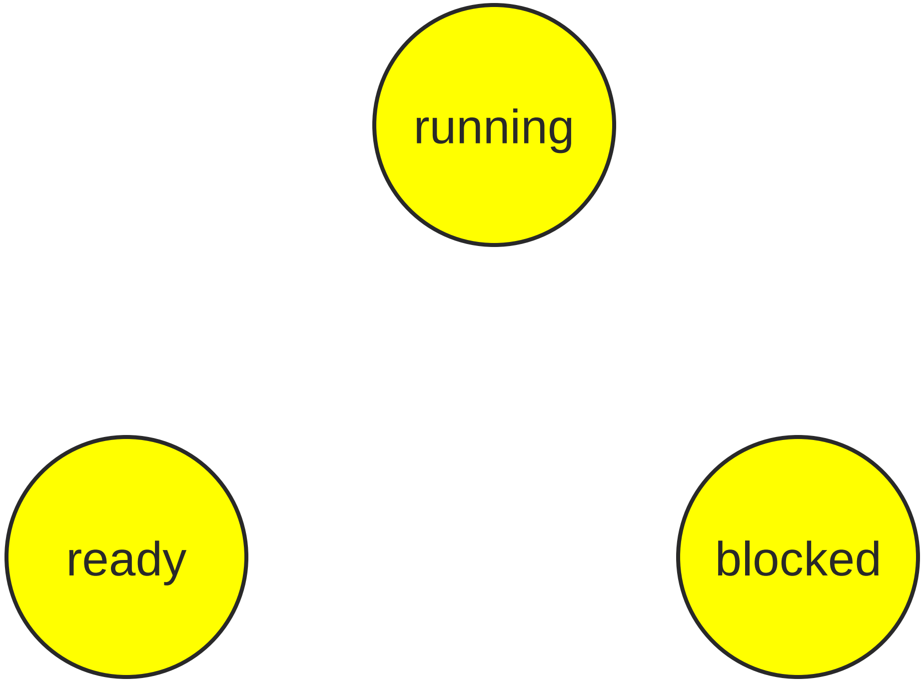
Five State Process Model

- "All models are wrong. Some Models are Useful"
  - George Box, Statistician
- 2 state model
  - Too simplistic
  - What does “Not Running” mean?
- 7 state model
  - Considers suspending process to disk
  - See Stallings 3.2

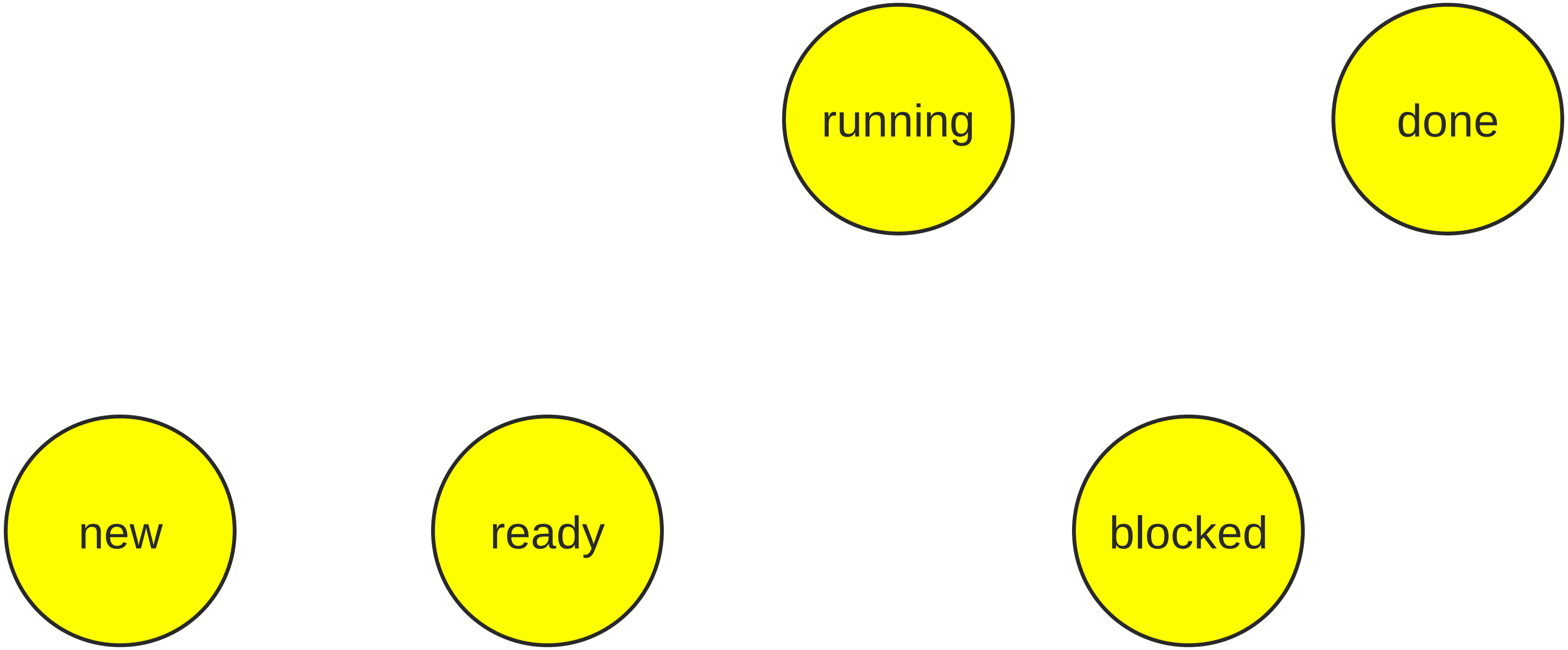
5 State Model - States



5 State Model - States



5 State Model - States

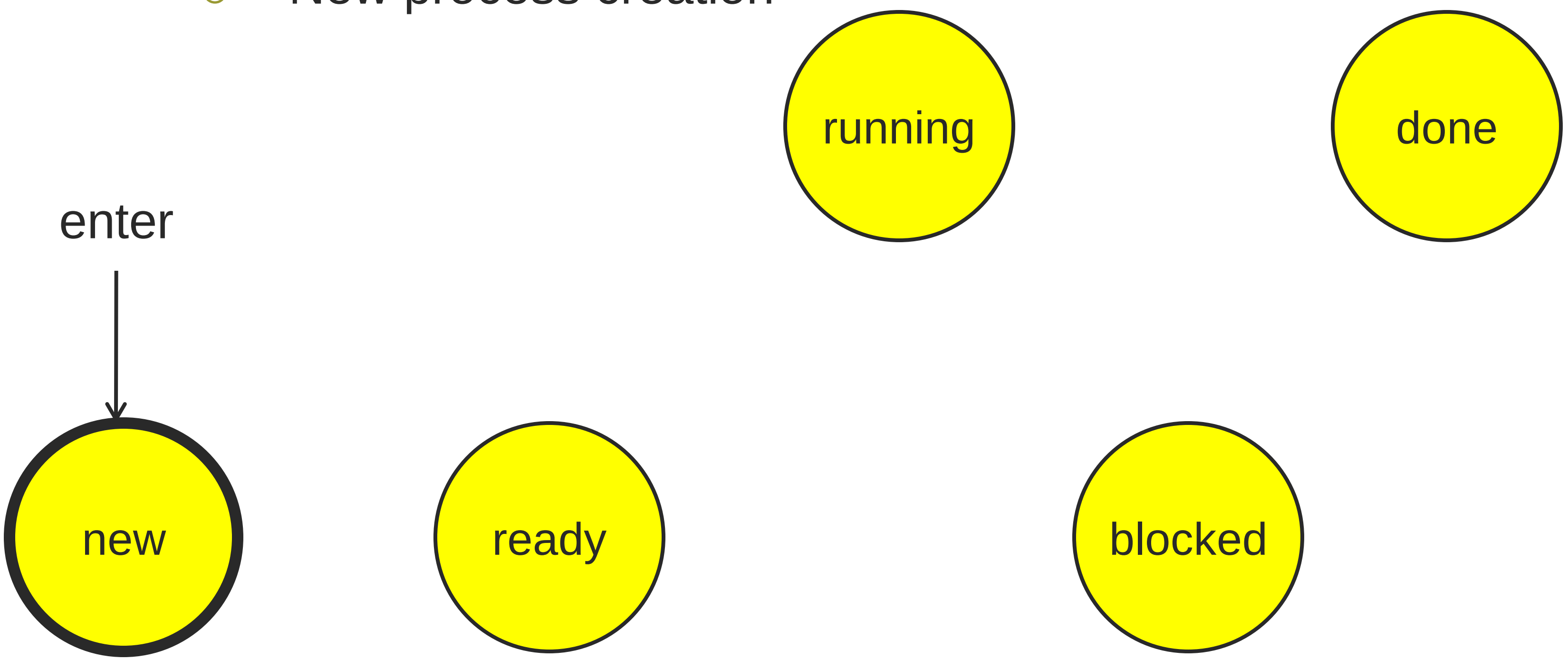


Five State Process Model

- Running
  - Currently executing
  - On a single processor machine, at most one process in the "running" state
- Ready
  - Prepared to execute
- Blocked
  - Waiting on some event
- New
  - Created, but not loaded into memory
- Done
  - Released from pool of executing processes

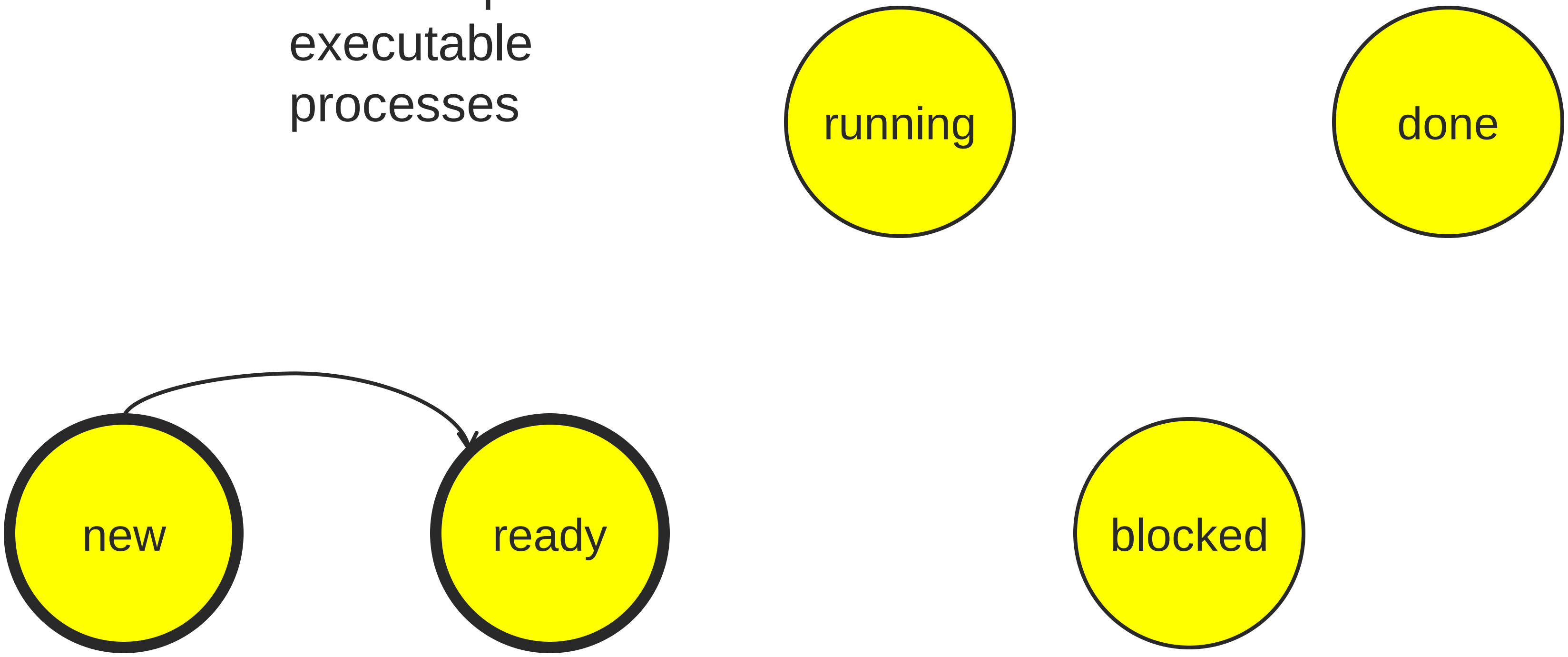
5 State Model - Transitions

- Null (nothing) to New
  - New process creation



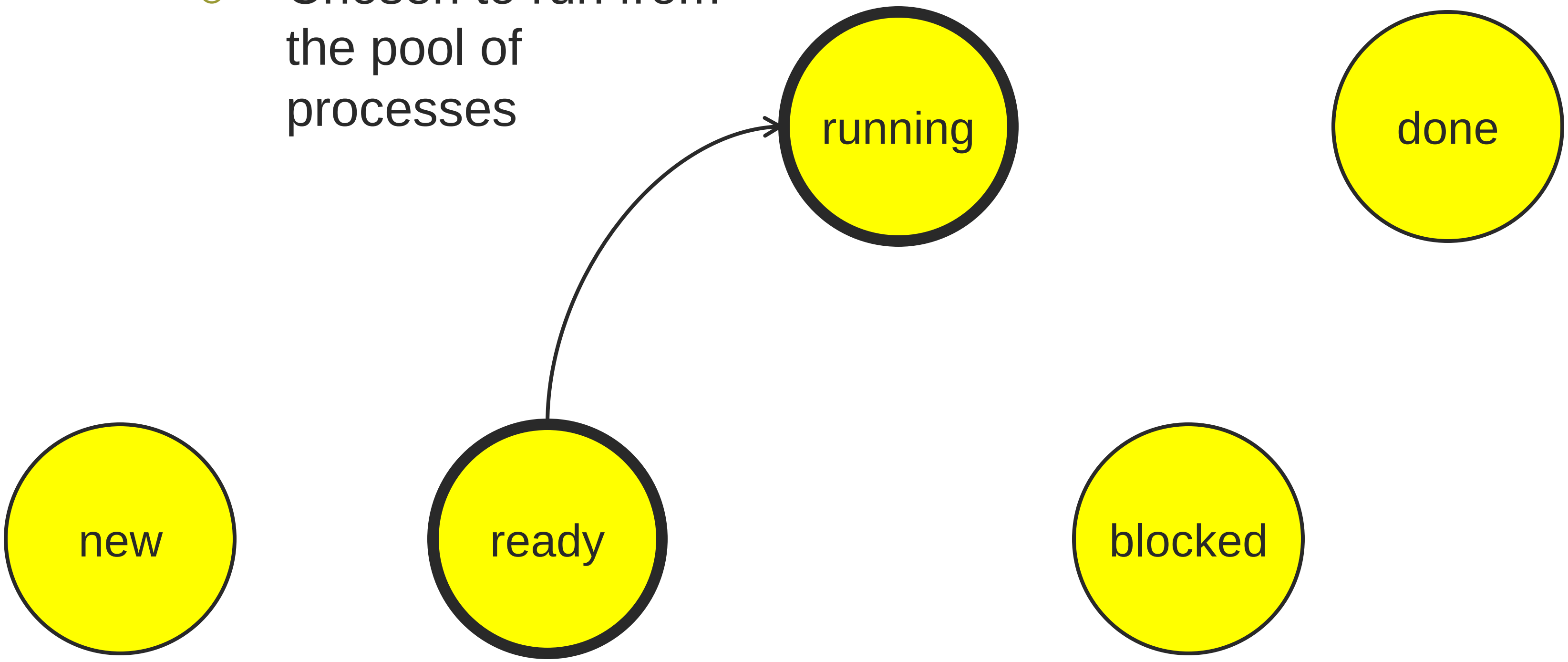
5 State Model - Transitions

- New to Ready
  - Move to pool of executable processes



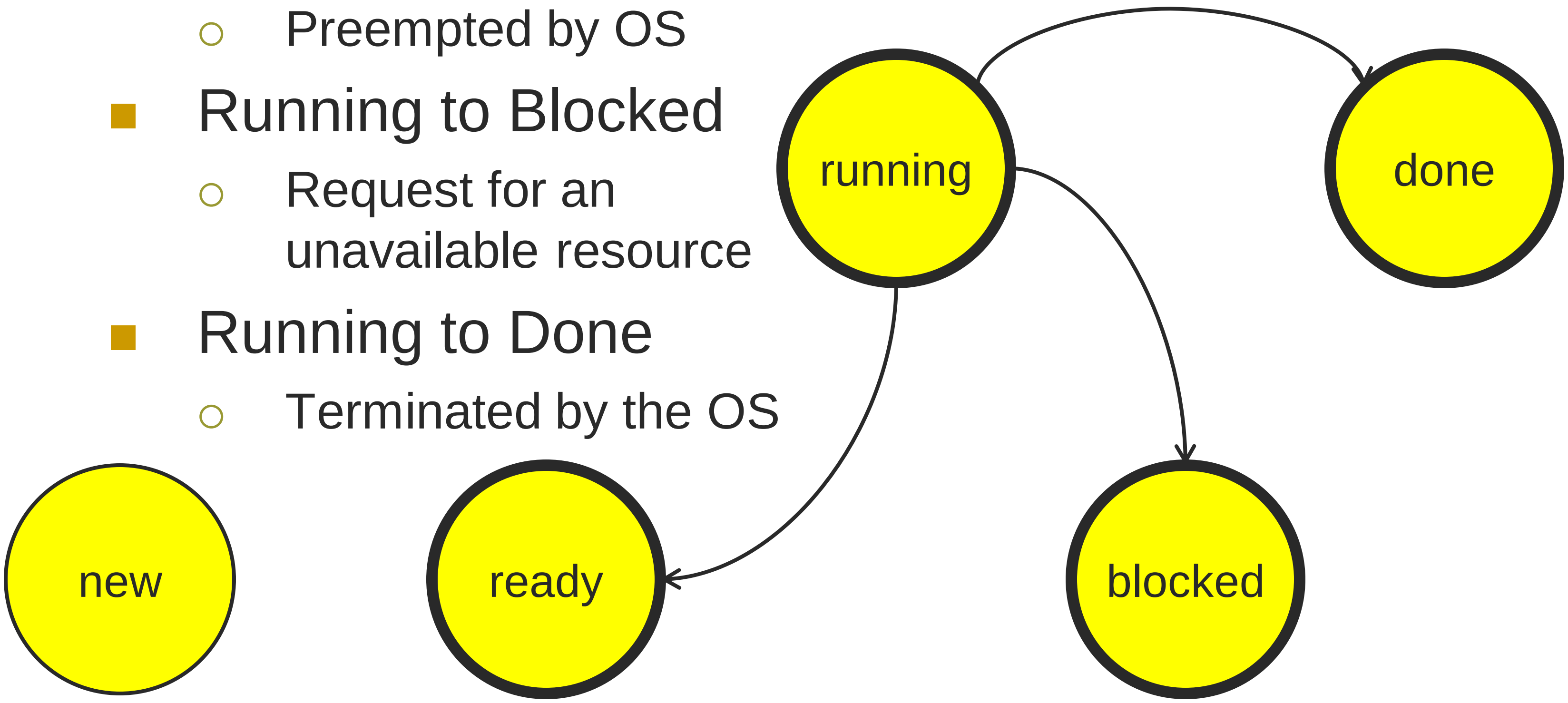
5 State Model - Transitions

- Ready to Running
  - Chosen to run from the pool of processes



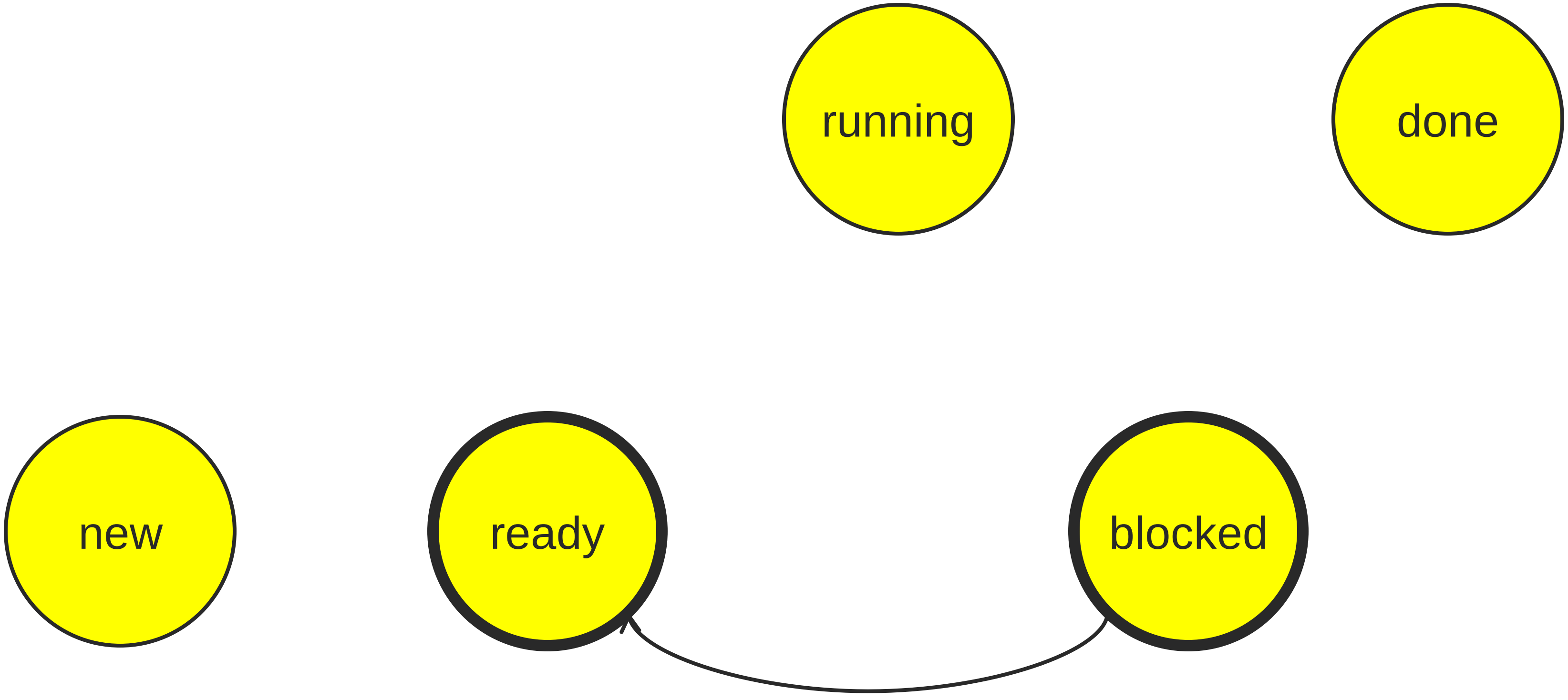
5 State Model - Transitions

- Running to Ready
  - Preempted by OS
- Running to Blocked
  - Request for an unavailable resource
- Running to Done
  - Terminated by the OS



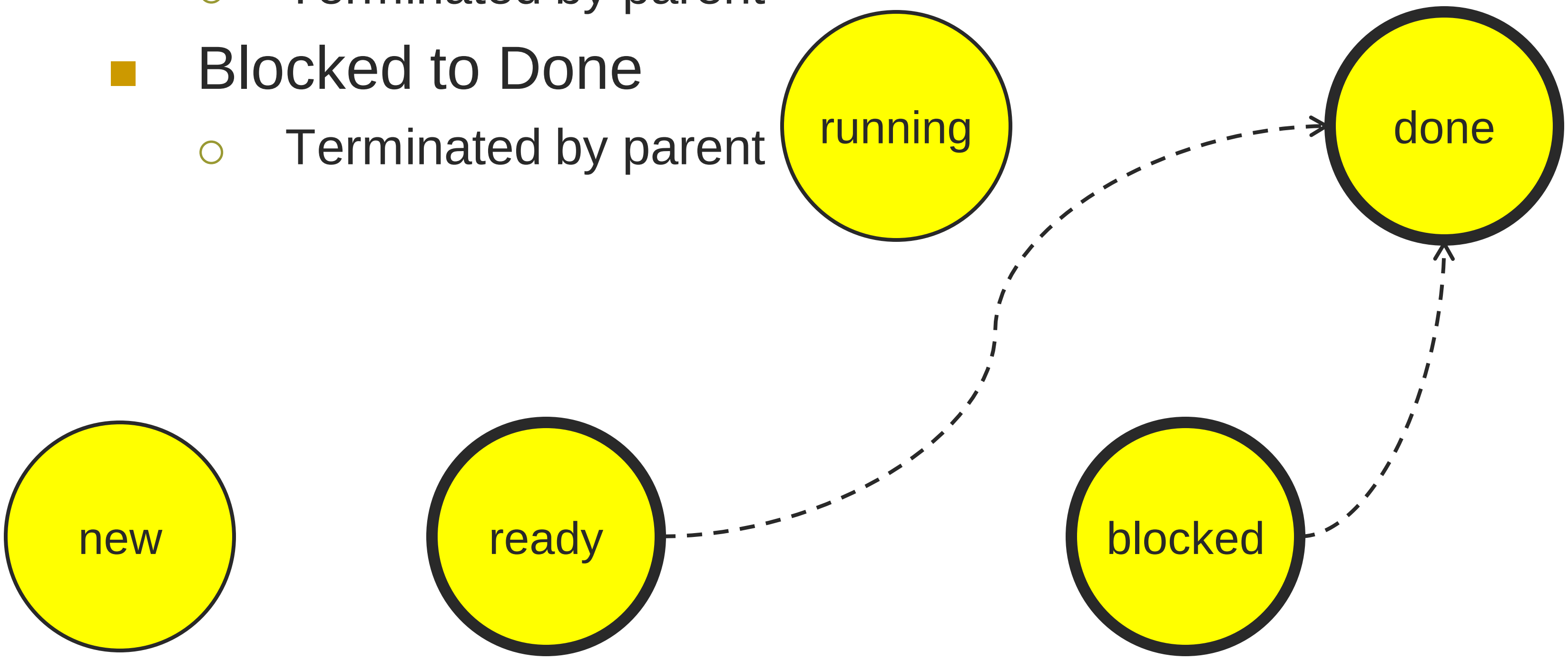
5 State Model - Transitions

- Blocked to Ready
  - Resource is now available

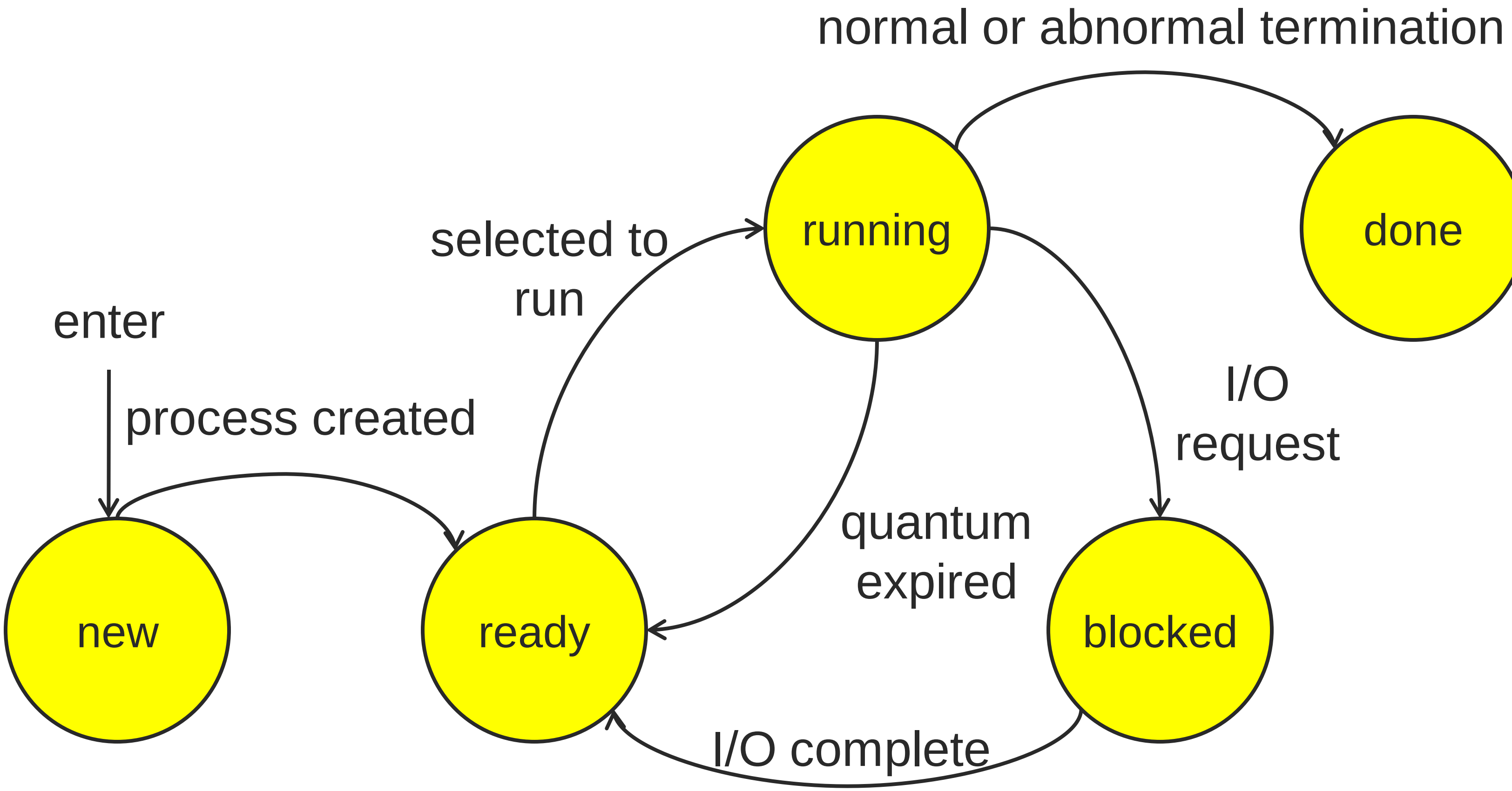


5 State Model - Transitions

- Ready to Done
  - Terminated by parent
- Blocked to Done
  - Terminated by parent

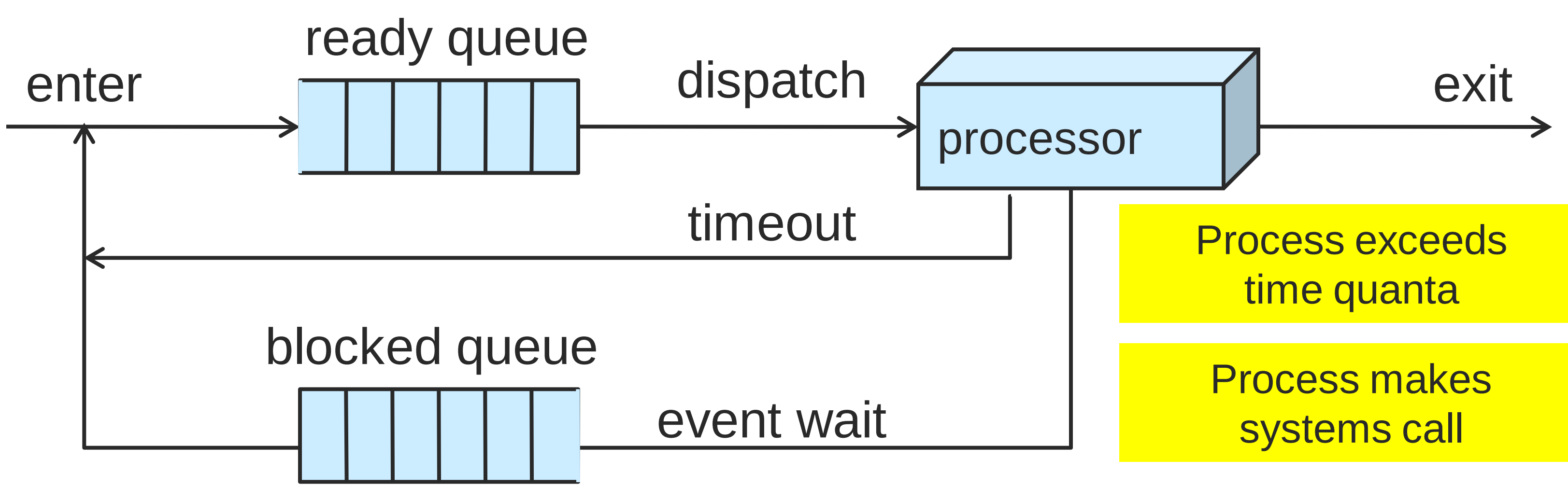


5 State Model - Transitions

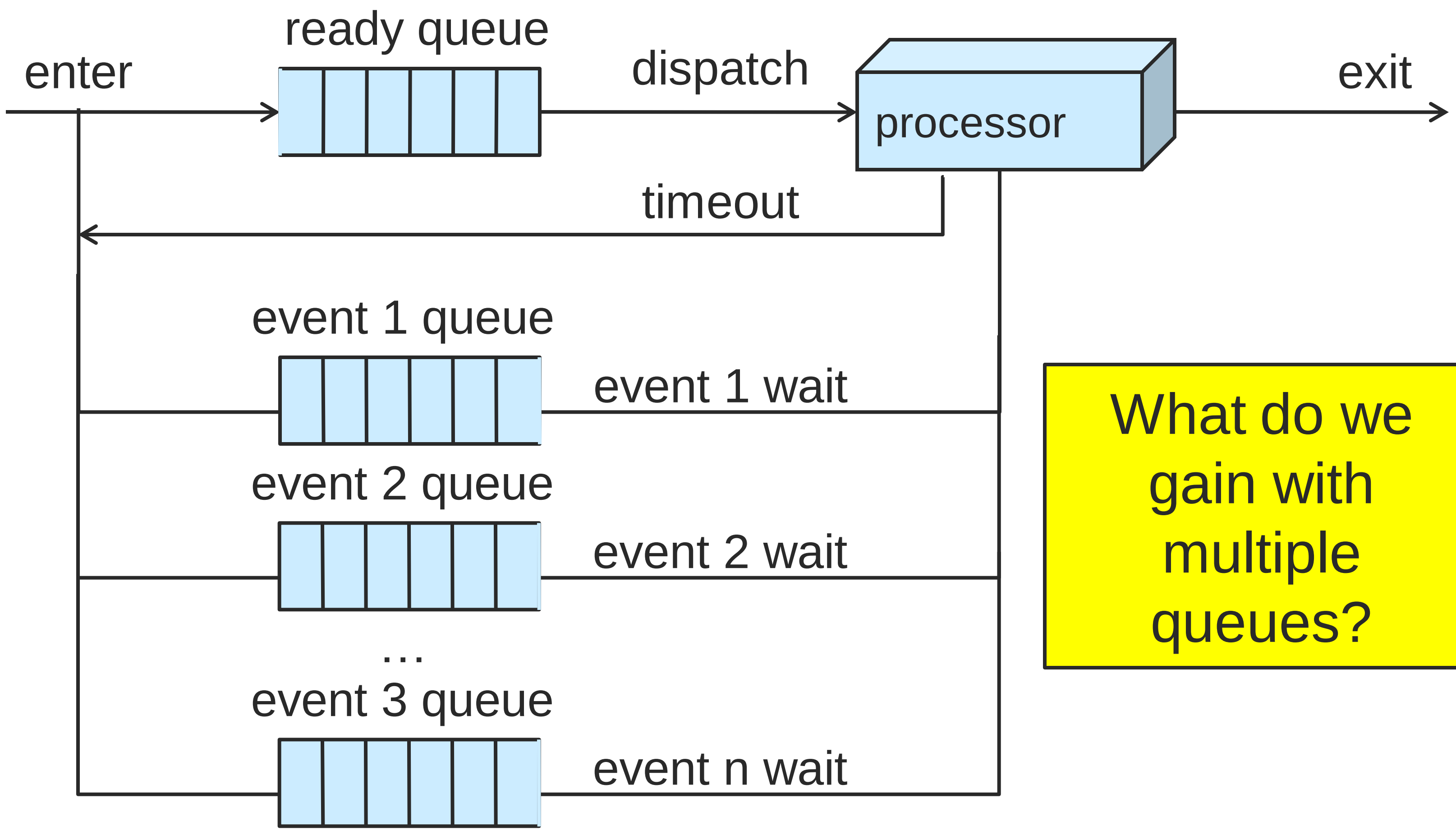


Process Queue Model

2 State Model: What is missing?

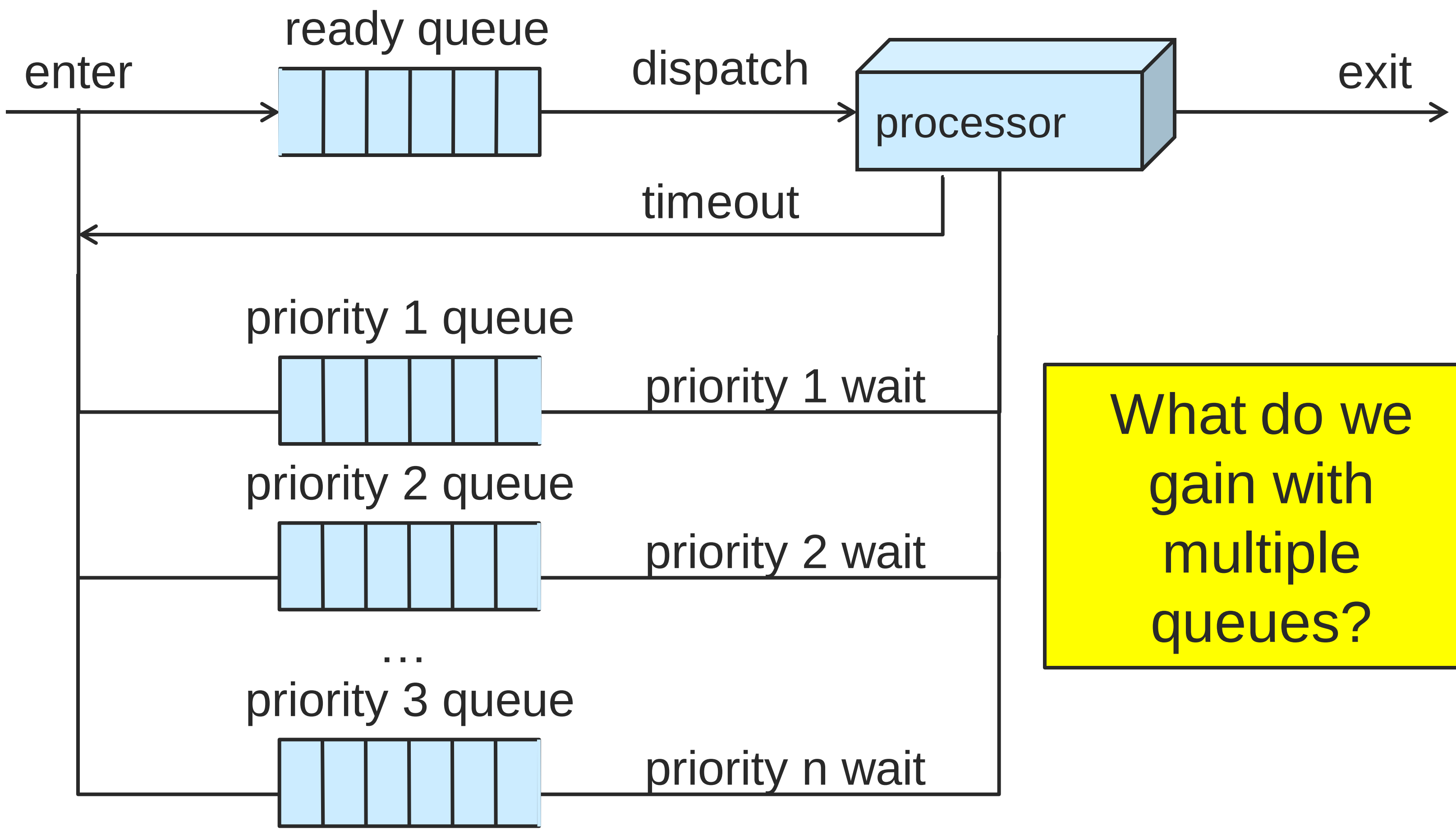


Process Queue Model



What do we gain with multiple queues?

Process Queue Model



What do we gain with multiple queues?

Orphans and Zombies

Orphans

- If the parent process dies no one is left to take care of the child
  - Child may consume large amounts of resources (CPU, File I/O)
  - Child Process is re-parented to the `init` process
    - `init` does not kill child but will wait for it.
    - child continues to run and run…

Zombies

- A Zombie is a child process that exited before it's parent called **`wait()`** to get the child's exit status
  - Does not consume many resources
    - Exit status (held in the program control block)
  - Also adopted by the **`init`** process
- Zombie Removal
  - Professional code installs signal handler (CS241 later lecture) for signal **`SIGCHLD`** which issues a **`wait()`** call

Take-away questions

- What would happen if user processes were allowed to disable interrupts?
- In a single CPU system what is the maximum number of processes that can be in the running state?
- Next: Threads and Thread Magic