

Operating Systems

MLFQ + Limited Direct Execution

Nipun Batra

MLFQ - Revision

MLFQ - Revision

- Rule 1: If $\text{Priority}(A) > \text{Priority}(B)$, A runs (B doesn't).
- Rule 2: If $\text{Priority}(A) = \text{Priority}(B)$, A & B run in RR.
- Rule 3: When a job enters the system, it is placed at the highest priority (the topmost queue).
- Rule 4: Once a job uses up its time allotment at a given level (regardless of how many times it has given up the CPU), its priority is reduced (i.e., it moves down one queue).
- Rule 5: After some time period S , move all the jobs in the system to the topmost queue.

Practice Session

```
./mlfq.py -s 5 -Q 2,10,15 -n 3 -j 3 -M 0 -m 30 -c
```

```
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS quantum length for queue 2 is 2
OPTIONS quantum length for queue 1 is 10
OPTIONS quantum length for queue 0 is 15
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False
```

For each job, three defining characteristics are given:

startTime : at what time does the job enter the system
runTime : the total CPU time needed by the job to finish
ioFreq : every ioFreq time units, the job issues an I/O
(the I/O takes ioTime units to complete)

Job List:

```
Job 0: startTime 0 - runTime 19 - ioFreq 0
Job 1: startTime 0 - runTime 24 - ioFreq 0
Job 2: startTime 0 - runTime 22 - ioFreq 0
```

Practice Session

```
./mlfq.py --jlist 0,40,0:20,20,0 -Q 5,10,10 -c
```

```
OPTIONS jobs 2
OPTIONS queues 3
OPTIONS quantum length for queue 2 is 5
OPTIONS quantum length for queue 1 is 10
OPTIONS quantum length for queue 0 is 10
OPTIONS boost 0
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False
```

For each job, three defining characteristics are given:

- startTime : at what time does the job enter the system
- runTime : the total CPU time needed by the job to finish
- ioFreq : every ioFreq time units, the job issues an I/O
(the I/O takes ioTime units to complete)

Job List:

```
Job 0: startTime 0 - runTime 40 - ioFreq 0
Job 1: startTime 20 - runTime 20 - ioFreq 0
```

Practice Session

```
./mlfq.py --jlist 0,40,0:10,20,0:20,30,0 -Q 5,10,10 -c -B 50
```

```
OPTIONS jobs 3
OPTIONS queues 3
OPTIONS quantum length for queue 2 is 5
OPTIONS quantum length for queue 1 is 10
OPTIONS quantum length for queue 0 is 10
OPTIONS boost 50
OPTIONS ioTime 5
OPTIONS stayAfterIO False
OPTIONS iobump False
```

For each job, three defining characteristics are given:

- startTime : at what time does the job enter the system
- runTime : the total CPU time needed by the job to finish
- ioFreq : every ioFreq time units, the job issues an I/O
(the I/O takes ioTime units to complete)

Job List:

```
Job 0: startTime 0 - runTime 40 - ioFreq 0
Job 1: startTime 10 - runTime 20 - ioFreq 0
Job 2: startTime 20 - runTime 30 - ioFreq 0
```

Practice Session

```
./mlfq.py -c
```

```
Here is the list of inputs:
```

```
OPTIONS jobs 3
```

```
OPTIONS queues 3
```

```
OPTIONS quantum length for queue 2 is 10
```

```
OPTIONS quantum length for queue 1 is 10
```

```
OPTIONS quantum length for queue 0 is 10
```

```
OPTIONS boost 0
```

```
OPTIONS ioTime 0
```

```
OPTIONS stayAfterIO False
```

```
For each job, three defining characteristics are given:
```

```
  startTime : at what time does the job enter the system
```

```
  runTime   : the total CPU time needed by the job to finish
```

```
  ioFreq    : every ioFreq time units, the job issues an I/O  
              (the I/O takes ioTime units to complete)
```

```
Job List:
```

```
Job 0: startTime 0 - runTime 84 - ioFreq 7
```

```
Job 1: startTime 0 - runTime 42 - ioFreq 2
```

```
Job 2: startTime 0 - runTime 51 - ioFreq 4
```


Practice Session

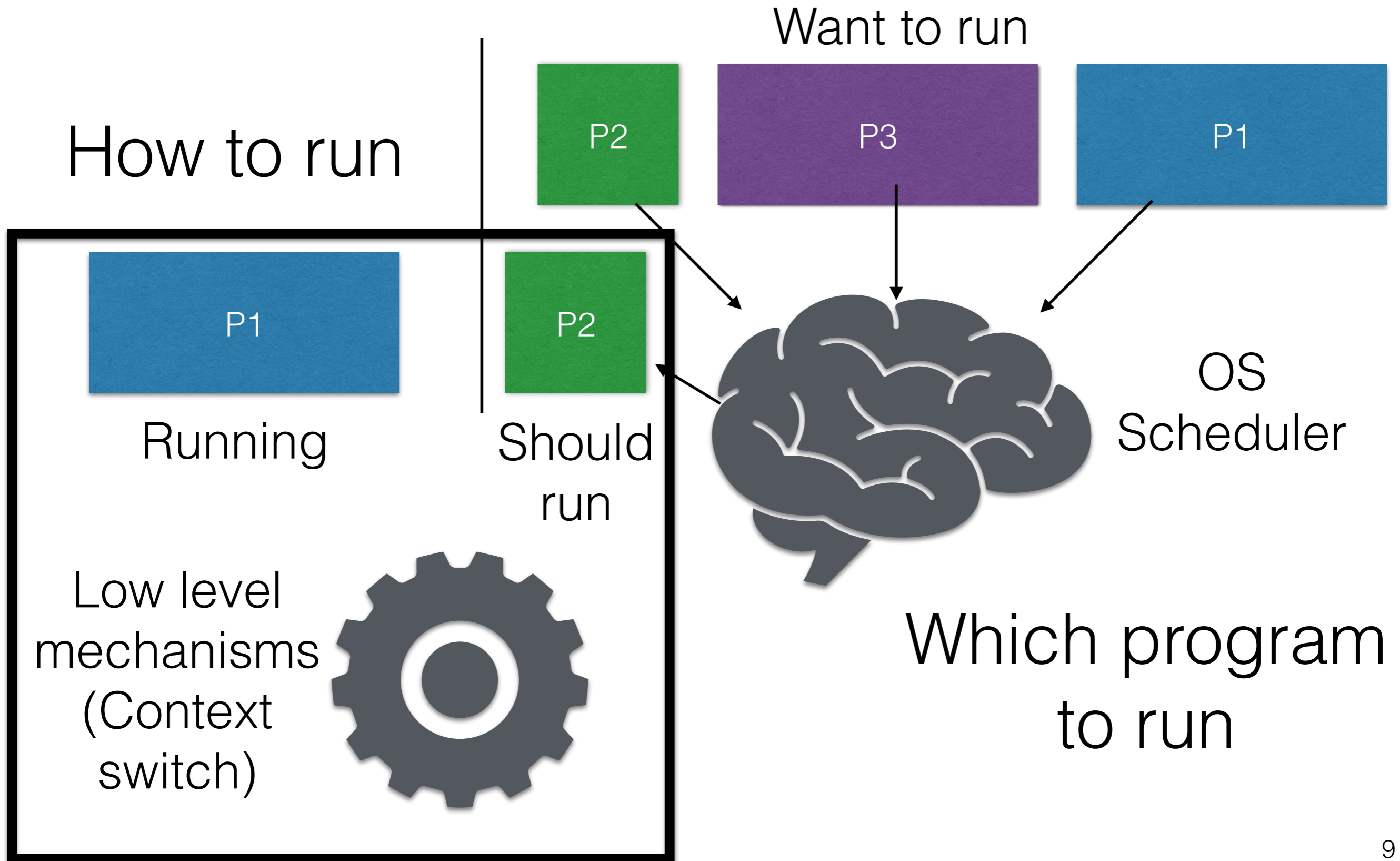
2. How would you run the scheduler to reproduce each of the examples in the chapter?
3. How would you configure the scheduler parameters to behave just like a round-robin scheduler?
4. Craft a workload with two jobs and scheduler parameters so that one job takes advantage of the older Rules 4a and 4b (turned on with the `-S` flag) to game the scheduler and obtain 99% of the CPU over a particular time interval.
5. Given a system with a quantum length of 10 ms in its highest queue, how often would you have to boost jobs back to the highest priority level (with the `-B` flag) in order to guarantee that a single long-running (and potentially-starving) job gets at least 5% of the CPU?
6. One question that arises in scheduling is which end of a queue to add a job that just finished I/O; the `-I` flag changes this behavior for this scheduling simulator. Play around with some workloads and see if you can see the effect of this flag.

More Topics

Will cover after remaining topics are finished:

1. Lottery scheduling
2. Multi-CPU scheduling (needs concurrency background)
3. Inter-process communication (IPC)

CPU Virtualisation Revisited



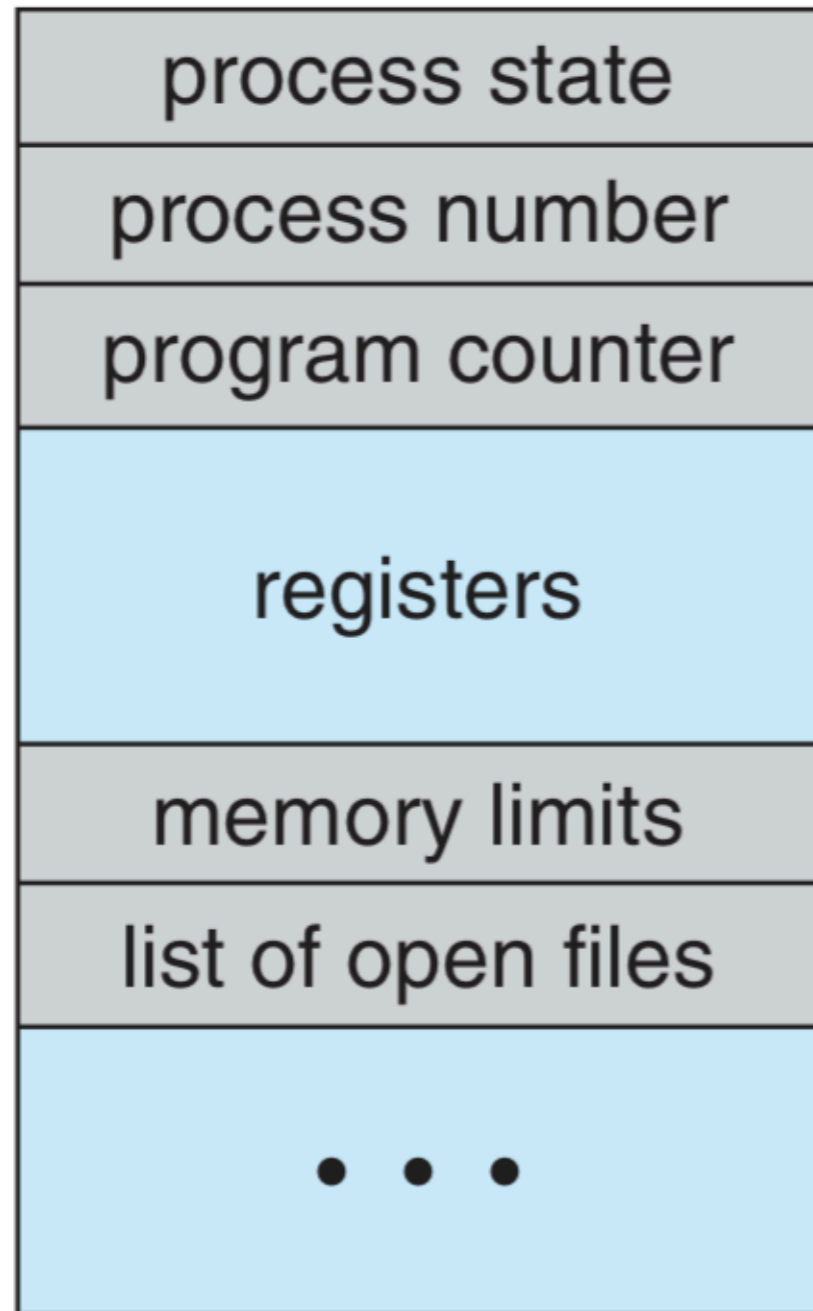
Process Control Block (PCB)

Process Control Block (PCB)

Data structure for process information

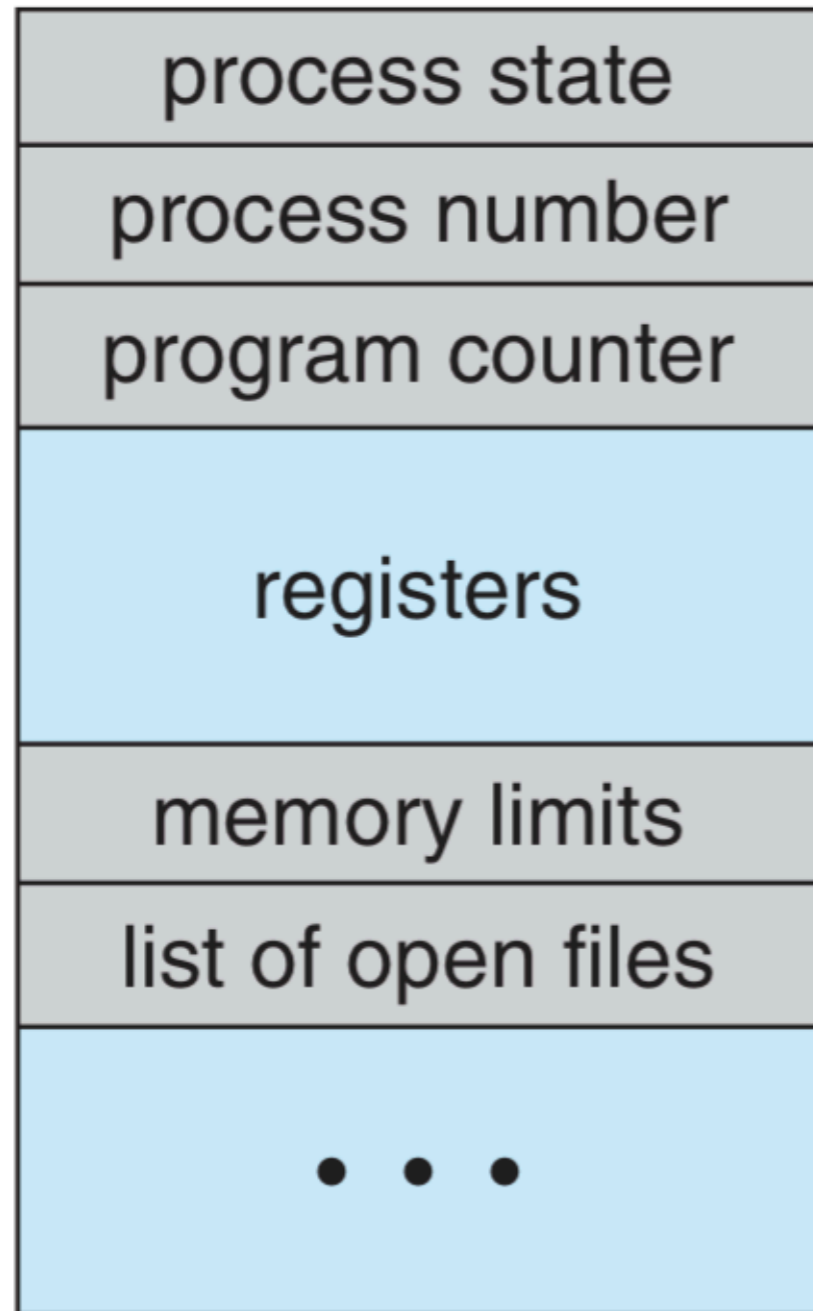
Process Control Block (PCB)

Data structure for process information



Process Control Block (PCB)

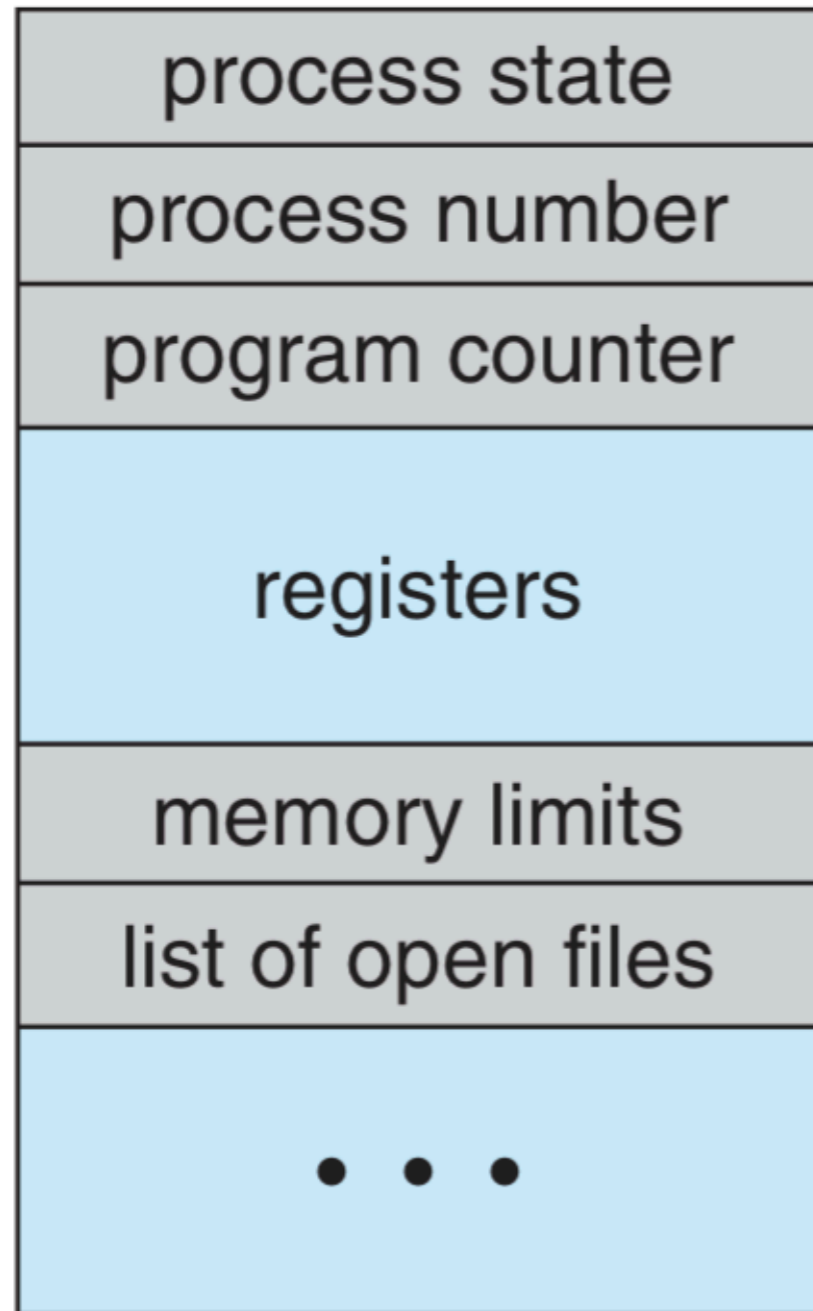
Data structure for process information



Running, Blocked, Ready

Process Control Block (PCB)

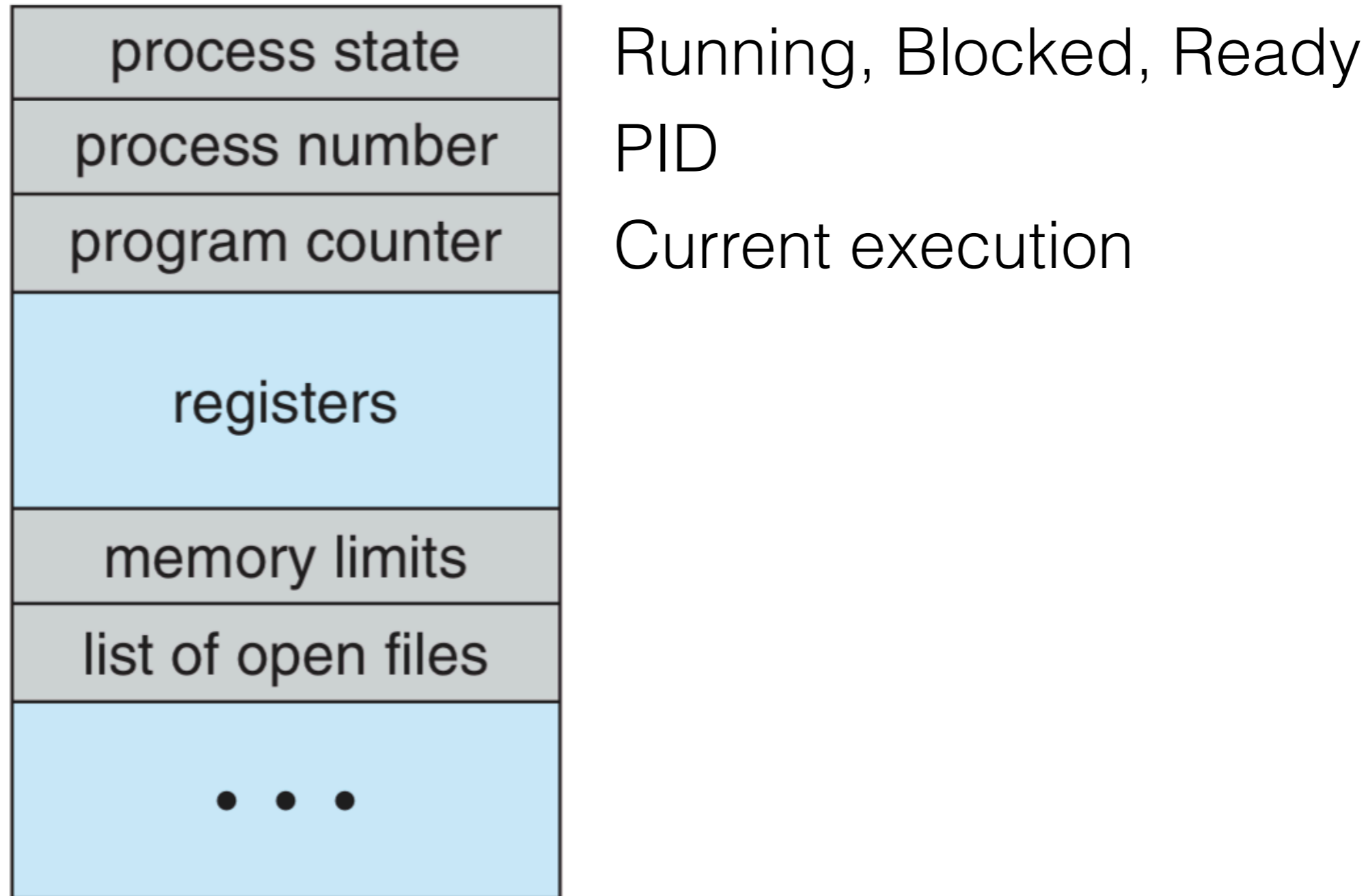
Data structure for process information



Running, Blocked, Ready
PID

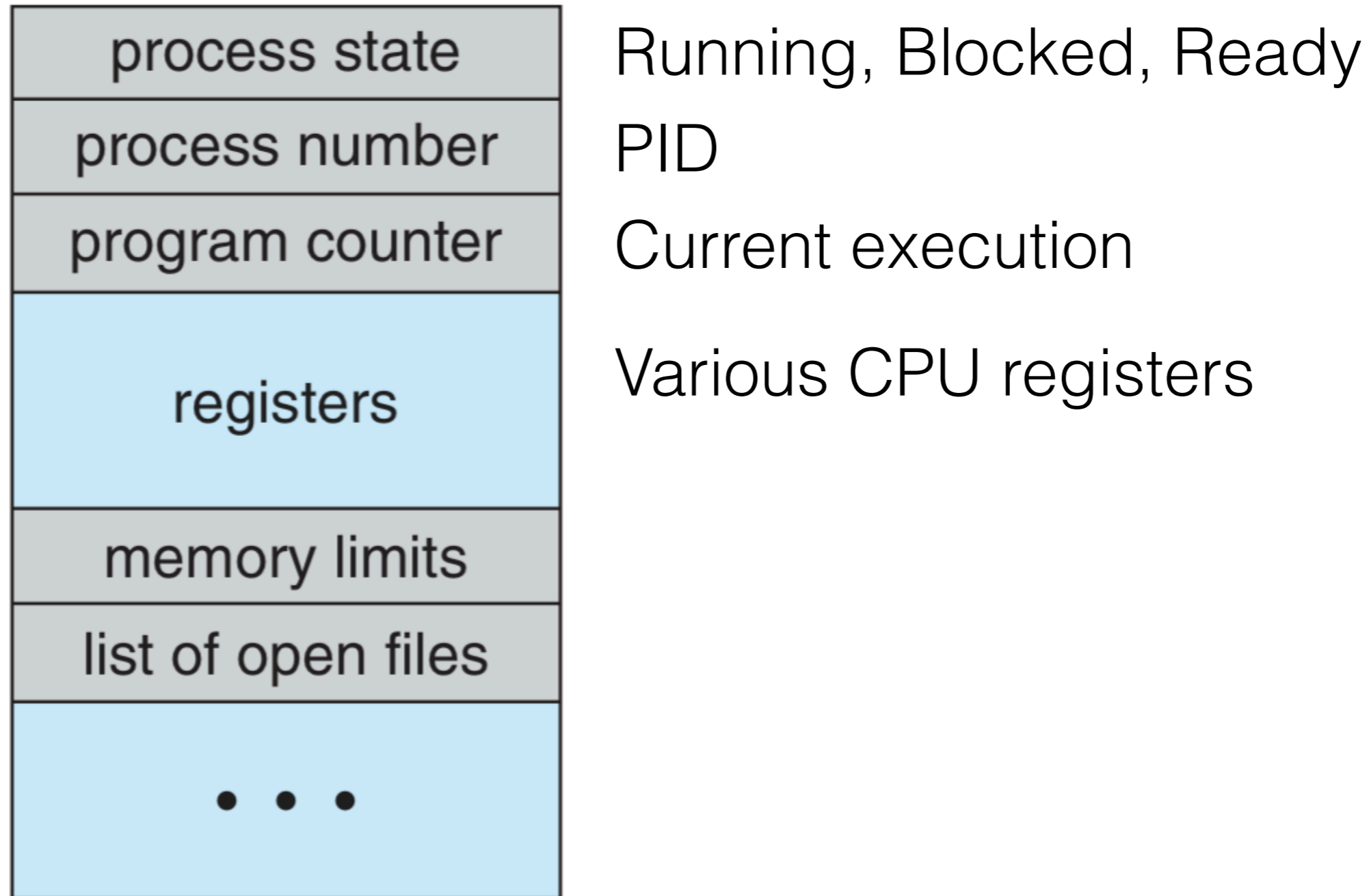
Process Control Block (PCB)

Data structure for process information



Process Control Block (PCB)

Data structure for process information



Process Control Block (PCB)

Data structure for process information

process state	Running, Blocked, Ready
process number	PID
program counter	Current execution
registers	Various CPU registers
memory limits	Management of page table etc.
list of open files	
• • •	

Process Control Block (PCB)

Data structure for process information

process state	Running, Blocked, Ready
process number	PID
program counter	Current execution
registers	Various CPU registers
memory limits	Management of page table etc.
list of open files	
• • •	CPU scheduling information: Priority, Run time, etc.

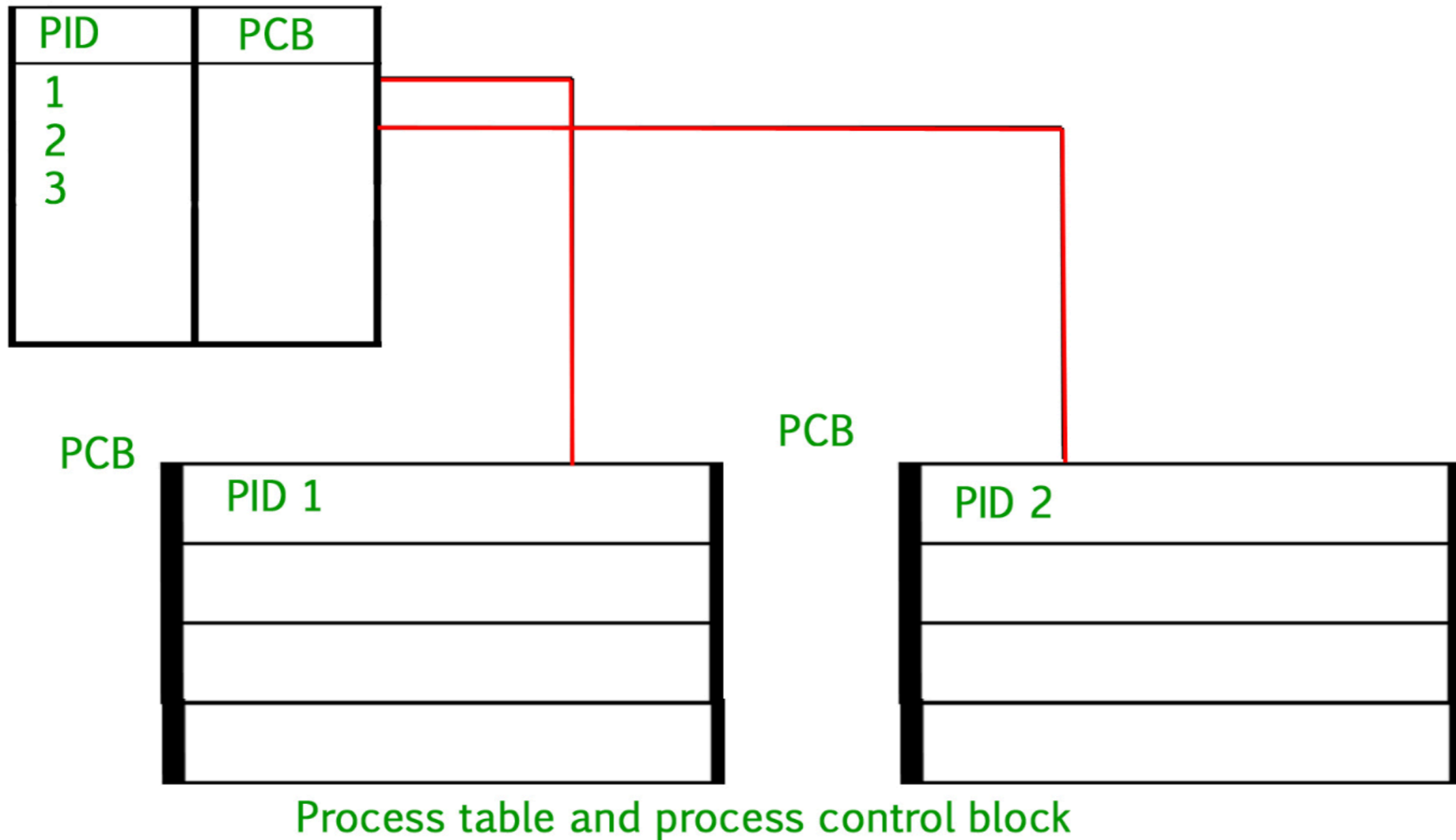
PCB in xv6

```
// the registers xv6 will save and restore
// to stop and subsequently restart a process
struct context {
    int eip;
    int esp;
    int ebx;
    int ecx;
    int edx;
    int esi;
    int edi;
    int ebp;
};

// the different states a process can be in
enum proc_state { UNUSED, EMBRYO, SLEEPING,
                  RUNNABLE, RUNNING, ZOMBIE };

// the information xv6 tracks about each process
// including its register context and state
struct proc {
    char *mem;                // Start of process memory
    uint sz;                  // Size of process memory
    char *kstack;            // Bottom of kernel stack
                              // for this process
    enum proc_state state;   // Process state
    int pid;                  // Process ID
    struct proc *parent;     // Parent process
    void *chan;              // If !zero, sleeping on chan
    int killed;              // If !zero, has been killed
    struct file *ofile[NOFILE]; // Open files
    struct inode *cwd;        // Current directory
    struct context context;   // Switch here to run process
    struct trapframe *tf;    // Trap frame for the
                              // current interrupt
};
```

Process Table



Could it be a linked list?

Core Challenges in CPU Virtualisation Mechanisms

Core Challenges in CPU Virtualisation Mechanisms

1. Performance - Minimise OS overhead

Core Challenges in CPU Virtualisation Mechanisms

1. Performance - Minimise OS overhead
2. Control - OS should maintain control

Core Challenges in CPU Virtualisation Mechanisms

1. Performance - Minimise OS overhead
2. Control - OS should maintain control
 1. Imagine OS schedules a process with infinite loop

Core Challenges in CPU Virtualisation Mechanisms

1. Performance - Minimise OS overhead
2. Control - OS should maintain control
 1. Imagine OS schedules a process with infinite loop
 2. We saw priority reduces over time in MLFQ

Direct Execution

OS

Program

Direct Execution

OS

Program

1. Create entry for process

Direct Execution

OS

Program

1. Create entry for process
2. Allocate memory for process

Direct Execution

OS

Program

1. Create entry for process
2. Allocate memory for process
3. Load program into memory

Direct Execution

OS

Program

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack

Direct Execution

OS

Program

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Program

1. Run main()

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

Program

1. Run main()
2. Execute return from main

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

1. Free memory

Program

1. Run main()
2. Execute return from main

Direct Execution

OS

1. Create entry for process
2. Allocate memory for process
3. Load program into memory
4. Set up stack
5. Execute call main()

1. Free memory
2. Remove process from process list

Program

1. Run main()
2. Execute return from main

Direct Execution Challenges

Direct Execution Challenges

1. How would OS stop the current process and run another

Direct Execution Challenges

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another
2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another

2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

Restricted Operations

1. How would OS stop the current process and run another

2. How does OS ensure that the program doesn't make illegal access (issuing I/O)



Restricted Operations

1. How would OS stop the current process and run another

2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

- Do we stop accessing I/O and network?

Restricted Operations

1. How would OS stop the current process and run another

2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

- Do we stop accessing I/O and network?

Restricted Operations

1. How would OS stop the current process and run another

2. How does OS ensure that the program doesn't make illegal access (issuing I/O)

- Do we stop accessing I/O and network?
- Goal: A process must be able to perform I/O and some other restricted operations, but without giving the process complete control over the system.

Restricted Operations

CPU Modes (hardware register to indicate)

Kernel Mode

User Mode

Restricted Operations

CPU Modes (hardware register to indicate)

Kernel Mode

User Mode

1. Restricted mode -
can not issue IO

Restricted Operations

CPU Modes (hardware register to indicate)

Kernel Mode

User Mode

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

Restricted Operations

CPU Modes (hardware register to indicate)

Kernel Mode

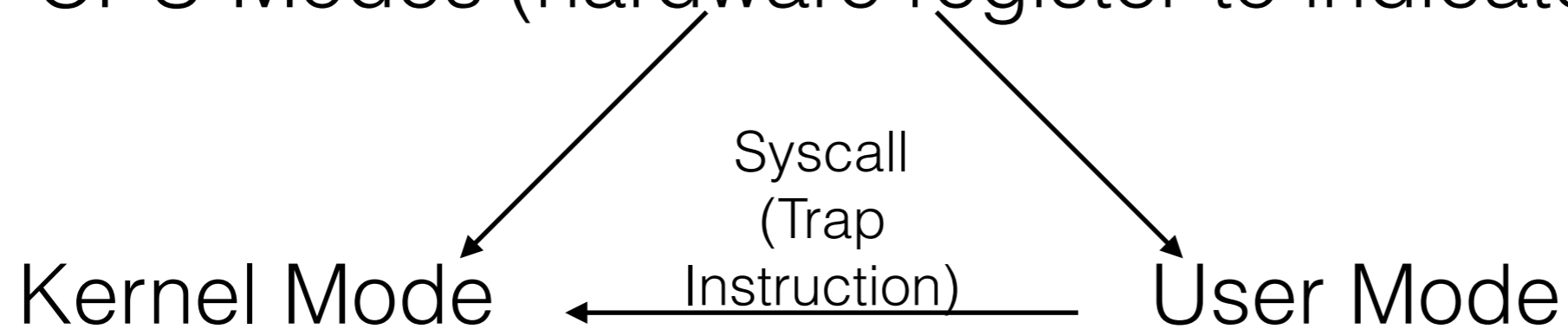
Code can issue IO.
Mode that OS runs in.

User Mode

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

Restricted Operations

CPU Modes (hardware register to indicate)



Code can issue IO.
Mode that OS runs in.

1. Restricted mode - can not issue IO
2. If tries to issue IO or restricted operation, exception raised

Kernel v/s User mode

- In **kernel mode**, the CPU has instructions to manage memory and how it can be accessed, plus the ability to access peripheral devices like disks and network cards. The CPU can also switch itself from one running program to another.
- In **user mode**, access to memory is limited to only some memory locations, and access to peripheral devices is denied. The ability to keep or relinquish the CPU is removed, and the CPU can be taken away from a program at any time.

Pop Quiz

Interrupt v/s Polling?