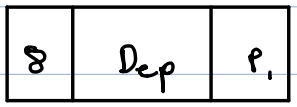
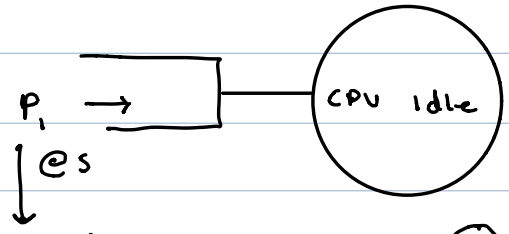
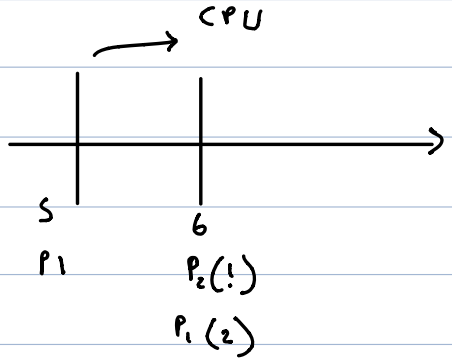
① FCFS

② SRTF



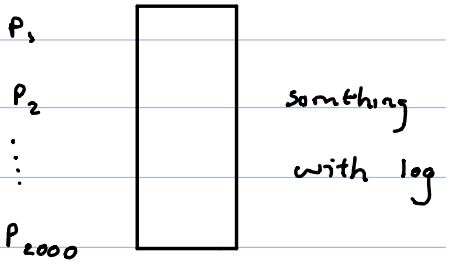


need 3 time at ⑥ P_2 time unit



$T_s = 6.06 + \mu = 16.166$

Average Service Time




```
while (!end_com) {
    event = get_event();
    clock = event.time;
    switch (event.type) {
        case ARR:
            processArrival(    )
        case DEP:
            processDeparture
```

Process_Arrival (time now, ...)

CPU Idle

→ schedule_event (Dep)

else

put in R.Q.

schedule_event (Arr, ...) ← next arrival

What is the event object

```
Event {
    time
    type_of_event
    which_kind
    pointer to the next event
}
```

Process Synchronization

- Control access to shared data (resource)
- Synchronization and coordination

counter = 4

P_0	P_1
counter ++	counter --
counter = 4	

P_0		P_1	
reg = counter;	1	reg = counter;	4
reg = reg + 1;	2	reg = reg + 1;	5
counter = reg;	3	counter = reg;	6

1 4 5 6 2 3 → counter = 5

1 4 2 5 3 6 → counter = 3

Race Condition

Multiple processes manipulate shared data, and the outcome depends on the order of execution

Critical Section Problem (CS)

Segment of code that manipulates shared data

* one process at a time inside the OS

P_0 or P_i

```
while (1) {
    entry-point();
    C.S.
```

solution

```
exit();
= Remainder section
};
```

Requirements for any solution :

1. Mutual Exclusion
2. Bounded Waiting (No process stores waiting to enter)
3. Progress Only process intrested in the critical section decide which one enters

How to control access to the C.S ?

1. Disable Interrupts

```
while (1) {                          ↙ local not on other CPU's
    disable_interrupts();
    C.S. ——————→ counter ++ / counter --
    enable_interrupts();
    R.S.
```

-> Only works on single CPU symptoms
- Degrade efficiency

2. Simple hardware instructions

- Atomic (Execute fully or not at all)

- Test and modify the contents, automically as a single instruction

[a] Test And Set (Returns Boolean)

atomic {

```
boolean TestAndSet (Boolean * target)
    boolean rv = *target
    *target = TRUE;
    return rv;
```

}

Disable interrupts is faster due to 3 lines executing.

```
lock = False;

P  do {
     while (TestAndSet (*lock))
        // do nothing
                                  Entry

     C.S.

     Lock = FALSE;                Exit

     R.S } while (1);
```

automic TAS

automic CAS