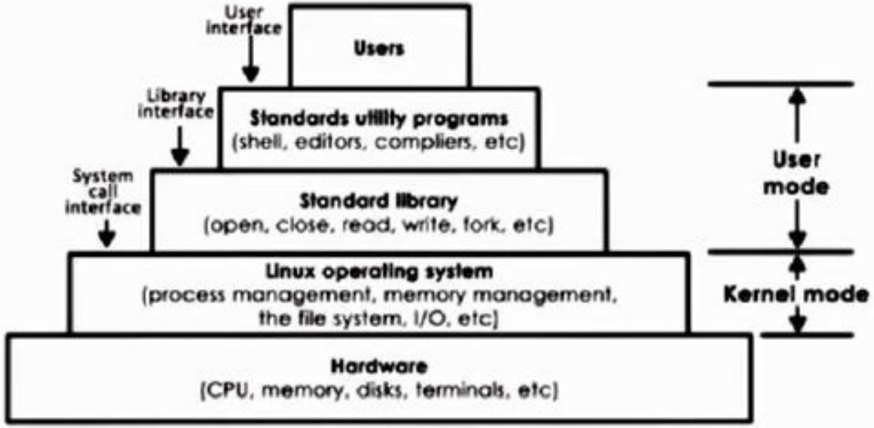
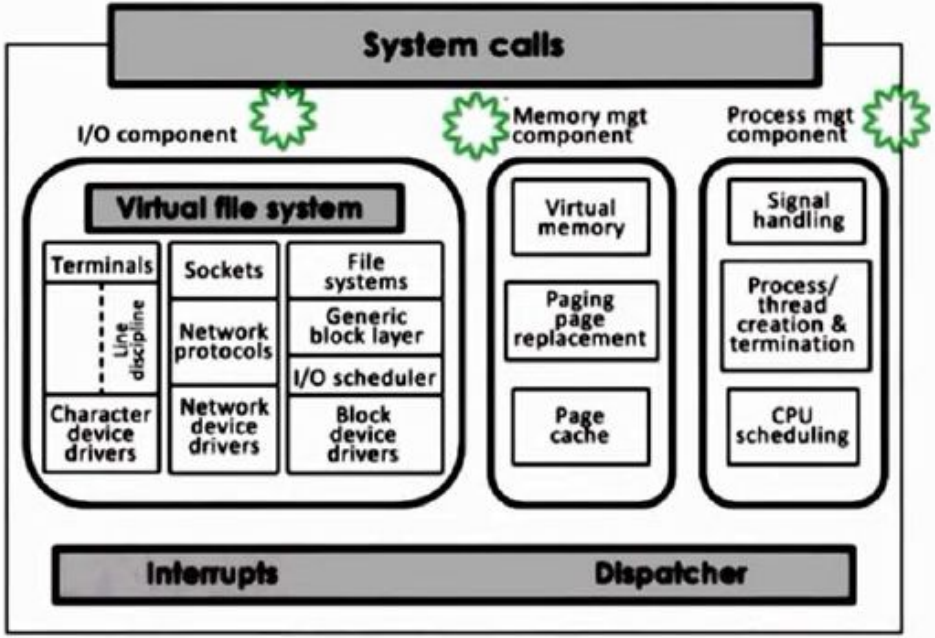
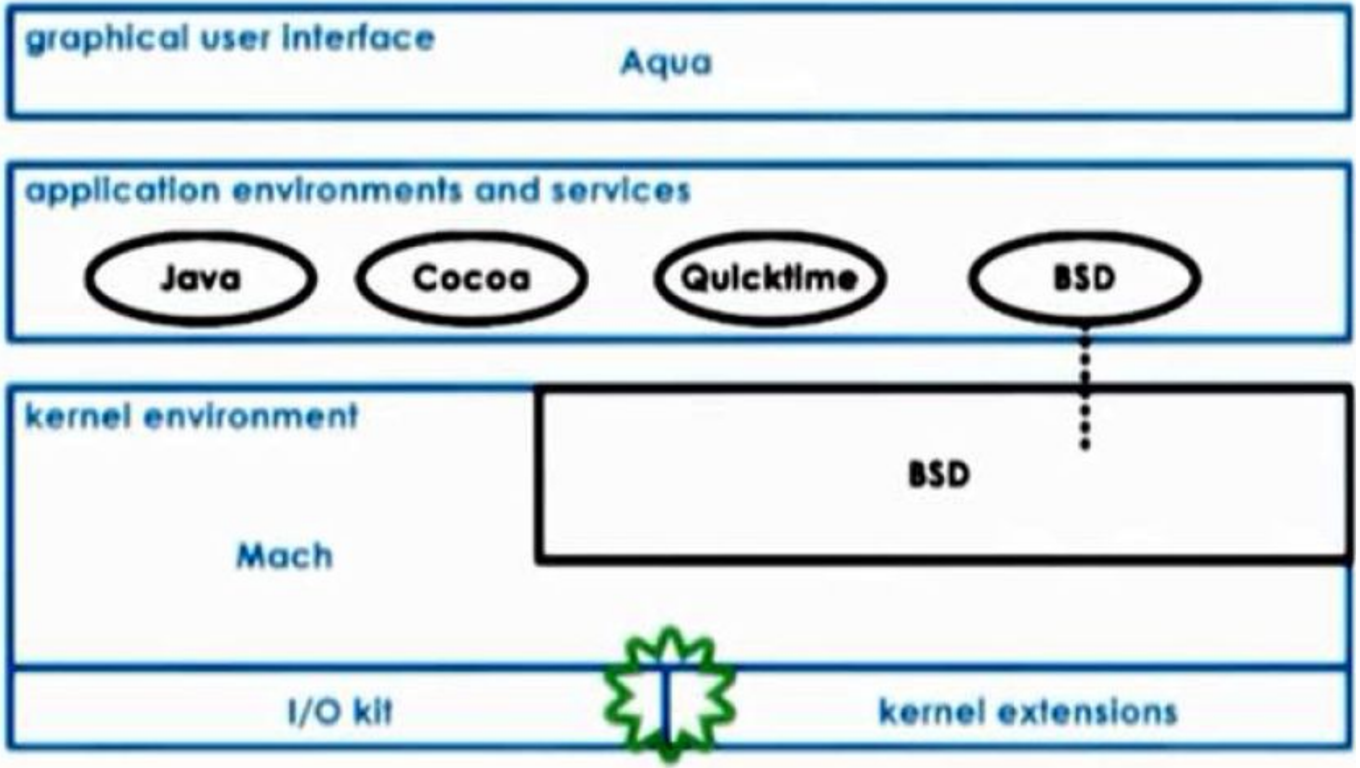
OS

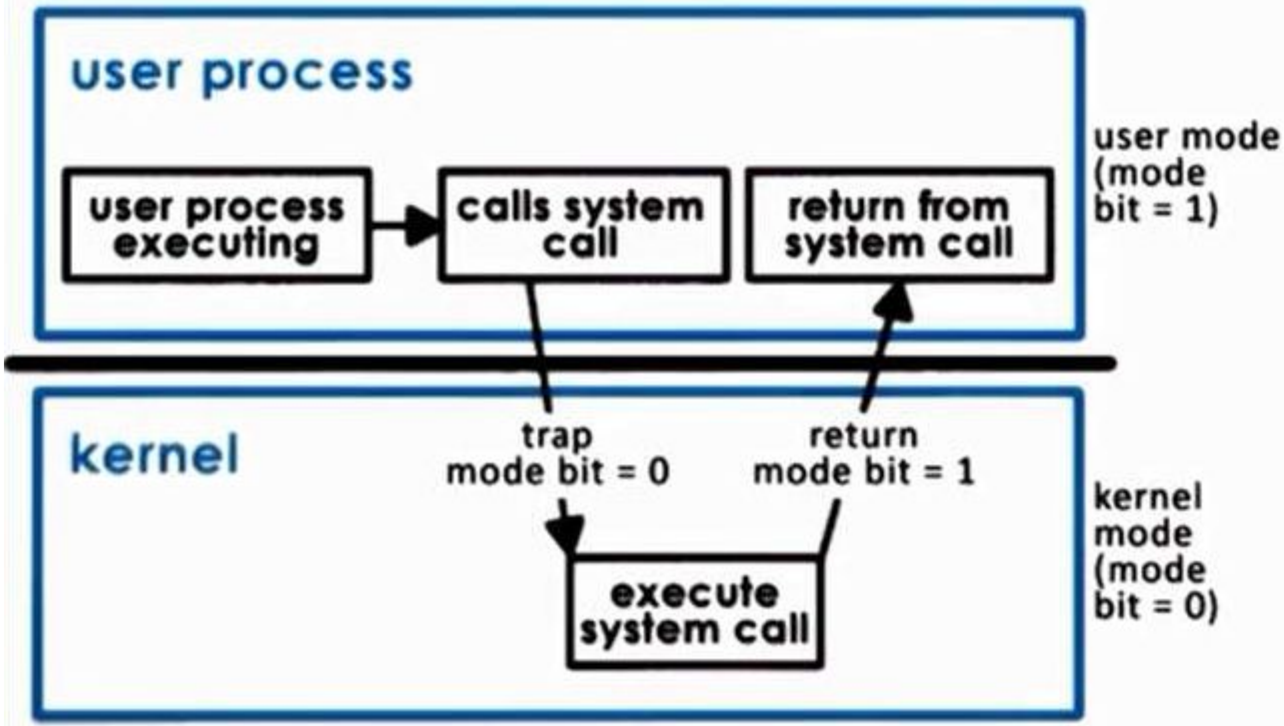
UCA 2024

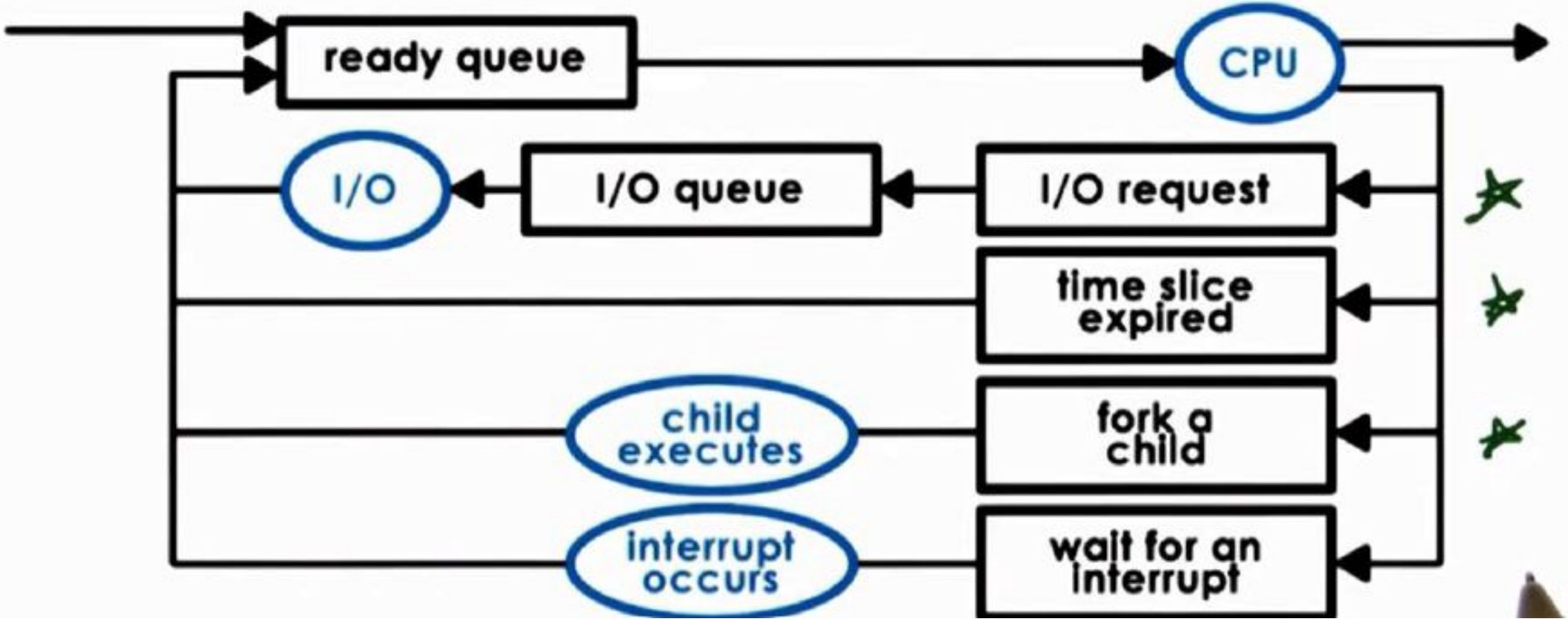

User interface
Users
Library interface
Standards utility programs
(shell, editors, compliers, etc)
System call interface
Standard library
(open, close, read, write, fork, etc)
Linux operating system
(process management, memory management, the file system, I/O, etc)
Hardware
(CPU, memory, disks, terminals, etc)
User mode
Kernel mode


System calls
I/O component
Memory mgt component
Process mgt component
Virtual file system
Terminals
Sockets
File systems
Line discipline
Network protocols
Generic block layer
I/O scheduler
Character device drivers
Network device drivers
Block device drivers
Virtual memory
Paging page replacement
Page cache
Signal handling
Process/ thread creation & termination
CPU scheduling
Interrupts
Dispatcher

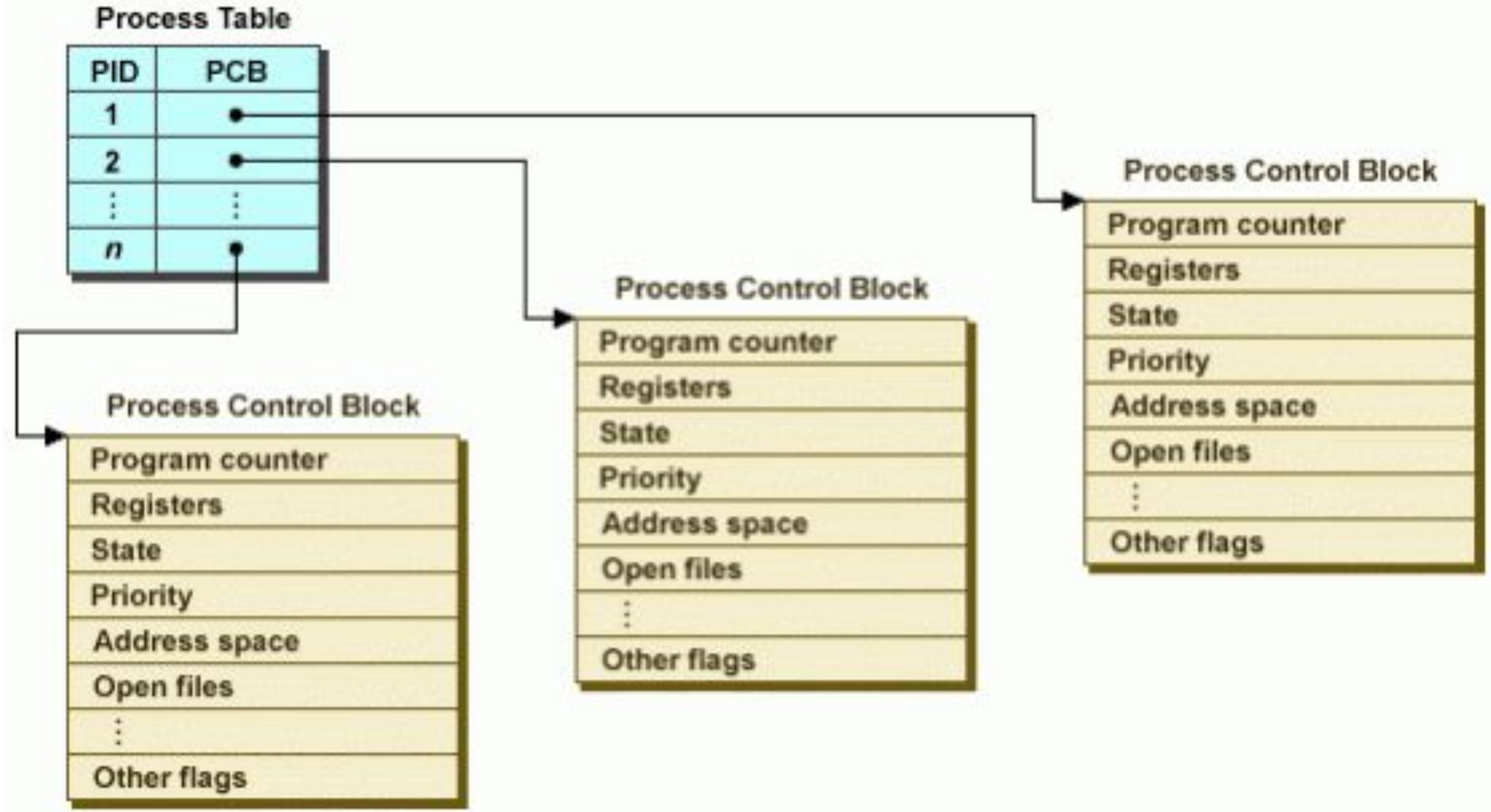
Mac OS Architecture




user process
user process executing
calls system call
return from system call
user mode (mode bit = 1)
kernel
trap mode bit = 0
return mode bit = 1
execute system call
kernel mode (mode bit = 0)


ready queue
CPU
I/O
I/O queue
I/O request
time slice expired
child executes
fork a child
interrupt occurs
wait for an interrupt

Process ID
Process state
Process priority
Accounting information
Program counter
CPU registers
PCB pointers
List of open files
Process I/O status information
....


Process Table
PID
PCB
1
2
n
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags
Process Control Block
Program counter
Registers
State
Priority
Address space
Open files
Other flags

31-Aug-2024

- Threads vs Processes
- Race condition demo using shared address space
- Locks - Mutex (exclusive access to one thread at a time) + critical section
- Producer consumer problem
- Condition variable :Conditional locks (specific condition before processing)
- Reader - Writer problem
- Spurious wakeups
- Deadlocks
- Multithreading patterns
    1. Boss-worker pattern
    2. Pipeline
    3. Layered

Shared Variables

```
Thread thread1;
Shared_list list;
Thread1 = fork(safe_insert, 4);
safe_insert(6);
join(thread1);
```

Producer consumer

```
for i = 0 → 10
        producers[i] = fork(safe_insert, Null);
consumer = fork(print_and_clear, Null);

//producer
lock(m) {
        list.insert(num);
}

//consumer
lock(m){
    if(list.isFull) print and clear list
}
```

Producer consumer using conditional variables

Conditional variables

- wait(mutex, val)
- signal(val)
- broadcast(val)

```
//producer
lock(m) {
      list.insert(num);
      if (list_isFull){
            signal(list_full);
      }
}
//consumer
lock(m){
    while(not list.isFull);
       —--> wait(m, list_full);  — c1, c2 ,c3
    Print and clear list;
}
```

Reader Writer Problem

```
// Reader
 lock(m){

 while(resource_count == -1){
        wait(m, read_ready)
 }
 resource_count++;
 }
 // read data
 lock(m){
 resource_count– -;
 if(resource_count == 0){
        signal(write_ready);
 }
```

```
//writer

       lock(m){
        while(resource_count != 0)
              wait(m, write_ready)
        resource_count = -1
       }
       //write data
       lock(m){
              resource_count = 0;

       }
       broadcast(read_ready)
       signal(write_ready);
```

Deadlocks

A cycle in wait graph ?

Deadlock prevention – Expensive
Deadlock detection – rollabck
Ostrich algorithm - Do not do anything

Boss worker

Boss assigns the job and worker does the job
Boss impacts throughput of system   1/boss_execution_time

Boss has to keep track of each worker
Handshaking with worker
Worker does not know about other workers

Boss doesn’t have to keep track of worker. It will just process queue.
Workers will pull a item from queue and process. step 1 and put in

Synchronize the queue - as a job should not be picked by multiple workers
What if the queue is empty at some point of time - boss is idle

Pattern is easy to implement but how to calculate no of workers for
efficient output. Boss does not know which task to assign to which

Pipeline pattern

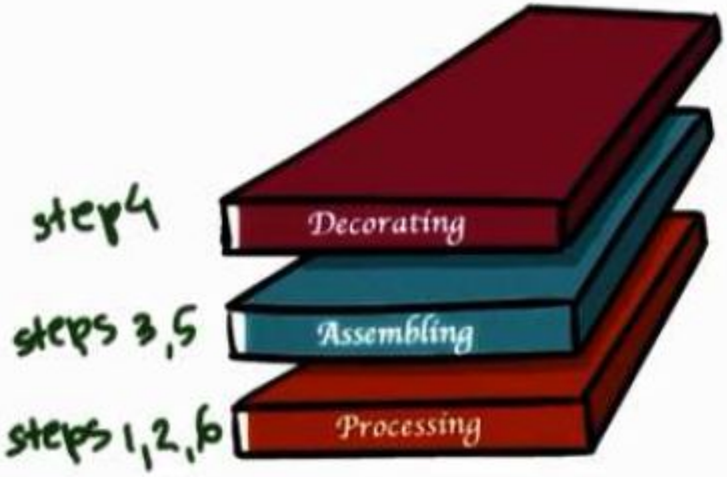
One thread one task
Multiple tasks concurrently in system
Throughput is weakest in herd
Shared buffer for communication between stages

Pros: specialization and locally
Cons: balancing and synchronization

Layers Pattern

Thread 1,2 for layer 1 : taks 1,2,3
Thread 3,4,5, for layer 2: task 4,5
And so on

Pros: specialization, less fine grained like pipeline
Cons : may not suitable for all applications, synchronization is still a concern



CPU Scheduling

- What is OS role as CPU scheduler
- Strategies (FCFS, SJF, Preemptive, Round Robin
- Metrics - Throughput, Avg wait time, Avg completion time
- O(1) linux and CFS
- CPU scheduling in multicore and multi CPU platforms
- Hyperthreading

What is CPU Scheduler

- Decides how and when process and Threads access CPU
- Including both Kernel and User level processes.
- Chooses one of the task from ready queue
- Runs when CPU is idle or new task become ready or time slice expires

Strategies:

- Assign task immediately
- Assign simple task first
- Assign complex task first

Metrics

- Throughput : number of task executed per second
- Average wait time : sum of wait time of all jobs/number of jobs
- Average completion time: sum of time to complete/no of jobs
- CPU utilization: total time CPU did productive work/total time

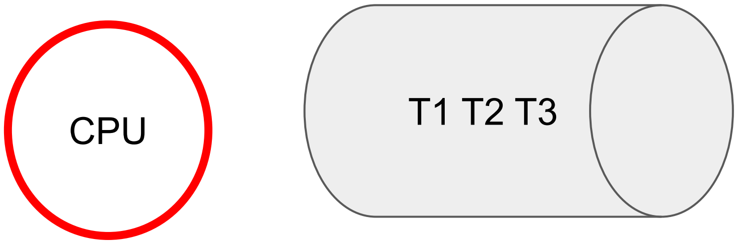
Assumption:

- Execution time is known
- No Preemption - run to completion
- Single CPU

FCFS

Task	Execution Time	Arrival Time
T1	1s	0
T2	10s	0
T3	1s	0

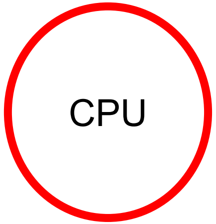
Throughput: 3/12
Avg wait time: (0+1+11)/3
Avg completion time: (1+11+12)/3

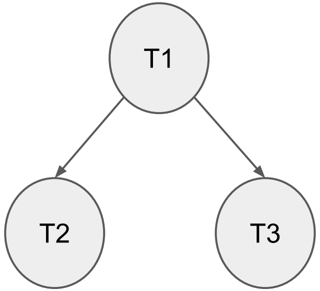


SJF

Task	Execution Time	Arrival Time
T1	1s	0
T2	10s	0
T3	1s	0

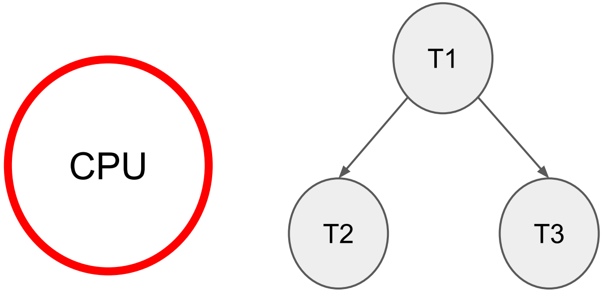
Throughput: 3/12
Avg wait time: (0+1+2)/3
Avg completion time: (1+2+12)/3





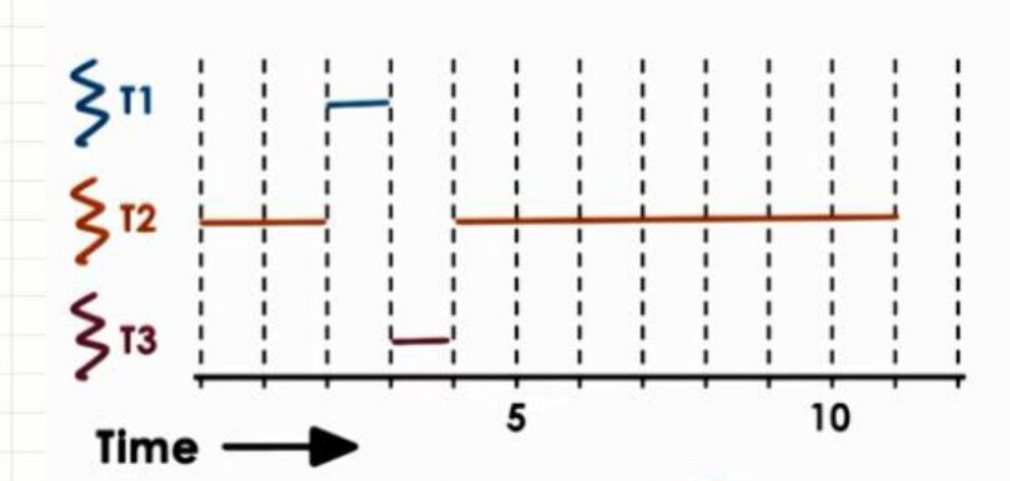
Preemptive SJF

Task	Execution Time	Arrival Time
T1	1s	2
T2	10s	0
T3	1s	2



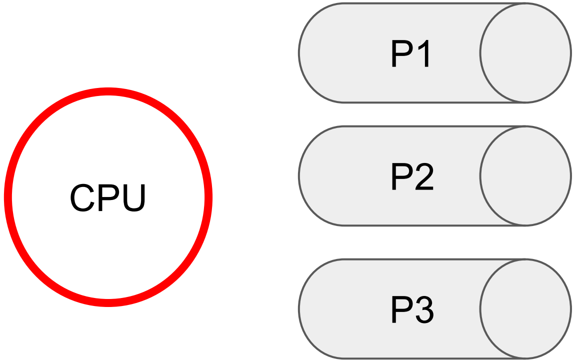
Throughput: 3/12
Avg wait time: (0+0+1)/3
Avg completion time: (1+2+12)/3

Good but not practical



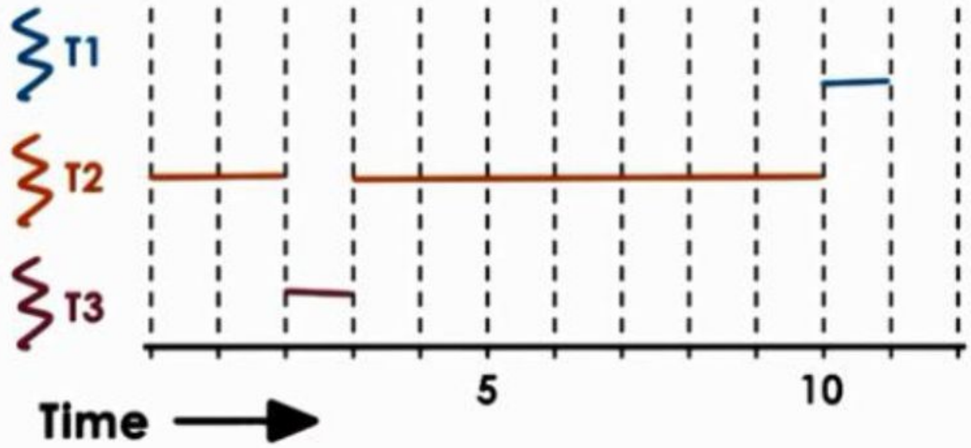
Preemptive with Priority

Task	Execution Time	Arrival Time	Priority
T1	1s	2	P1
T2	10s	0	P2
T3	1s	2	P3



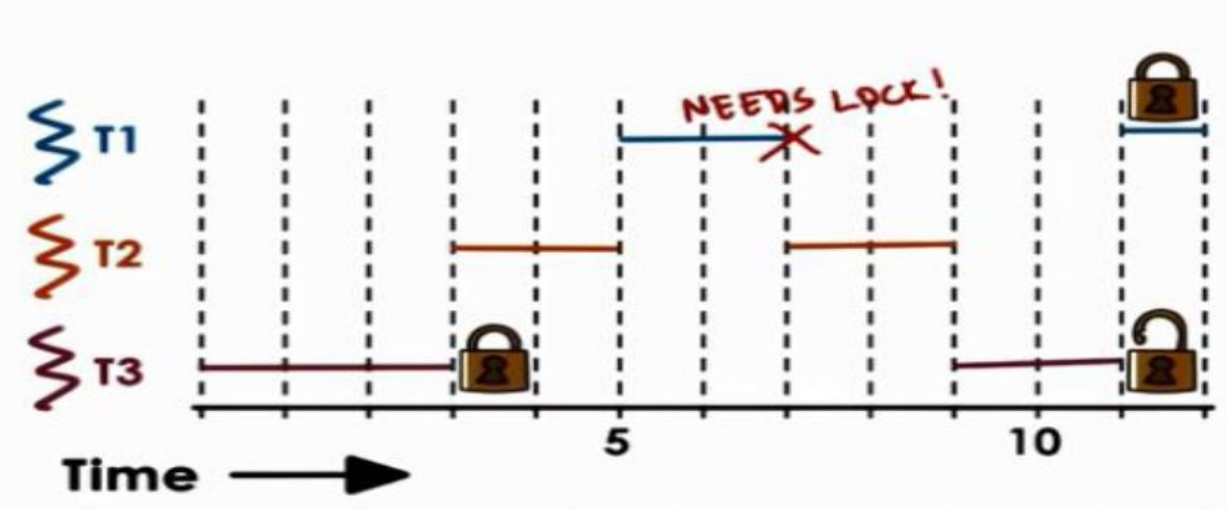
Throughput: 3/12
Avg wait time: (10+0+0)/3
Avg completion time: (9+1+11)/3

Starvation!!
Priority = (Actual priority+time spend in ready queue)



Priority Inversion

Task	Execution Time	Arrival Time	Priority
T1	3s	5	P3
T2	4s	3	P2
T3	90s	0	P1

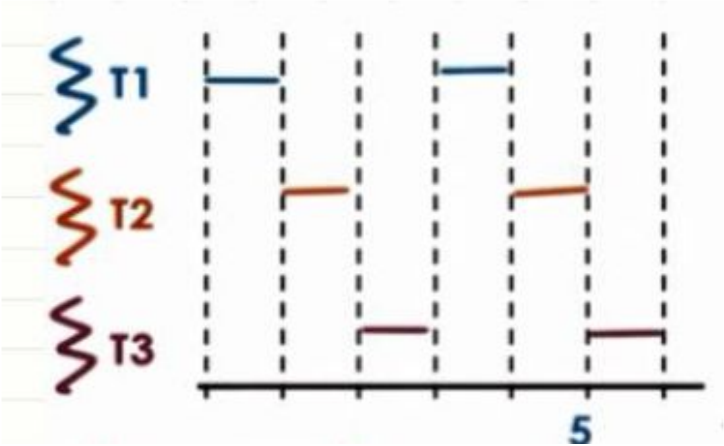


Boost Priority

- Mutex owner has max priority
- Decrease again on lock release

Round Robin

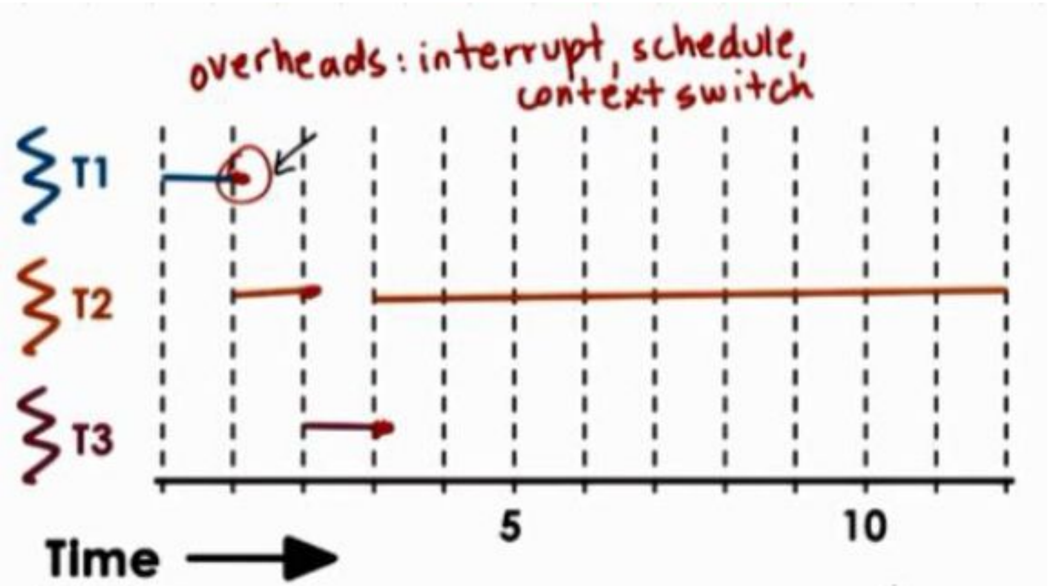
Task	Execution Time	Arrival Time
T1	2	0
T2	2	0
T3	2	0



- time slice/quantum = 1
- Max amount of uninterrupted time given to a task
- IO task may run for less time
- CPU bound task will be preempted after time slice
- Run queue can still support priority

Round Robin with Time Slicing

Task	Execution Time	Arrival Time
T1	1s	0
T2	10s	0
T3	1s	0



Throughput: 3/12
Avg wait time: (0+1+2)/3
Avg completion time: (1+12+3)/3

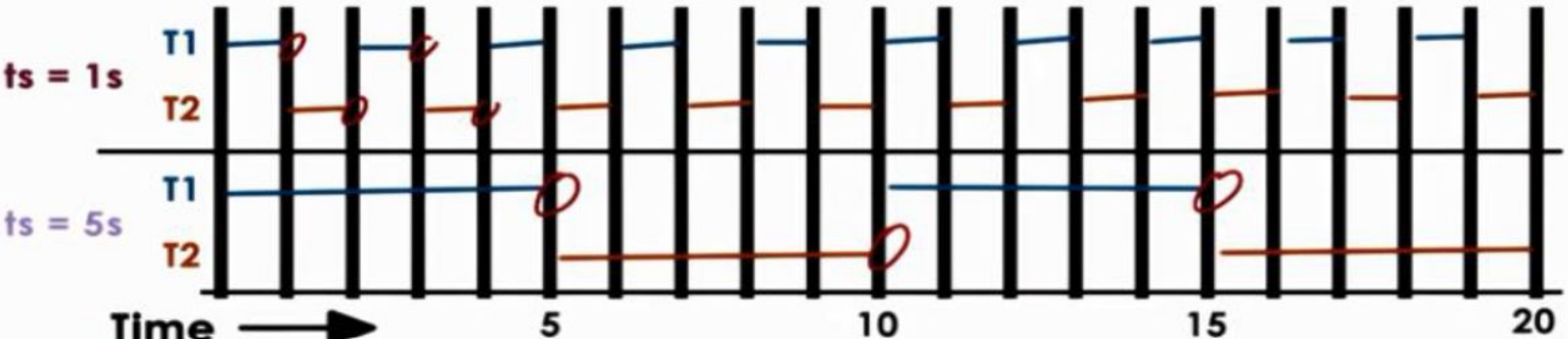
- Short task finishes first
- Seems more responsive
- IO operation initiated sooner
- But we assumed time slice (ts) >>>> context switch time

Time Slicing

2 CPU Bound task with execution time = 10s each
Context switch time = 0.1s

Ex: ts=1
Throughput: 2/(20+19x.1)
Avg wait time: (0+1.1)/2
Avg completion time: (19+20+1.8+1.9)/2

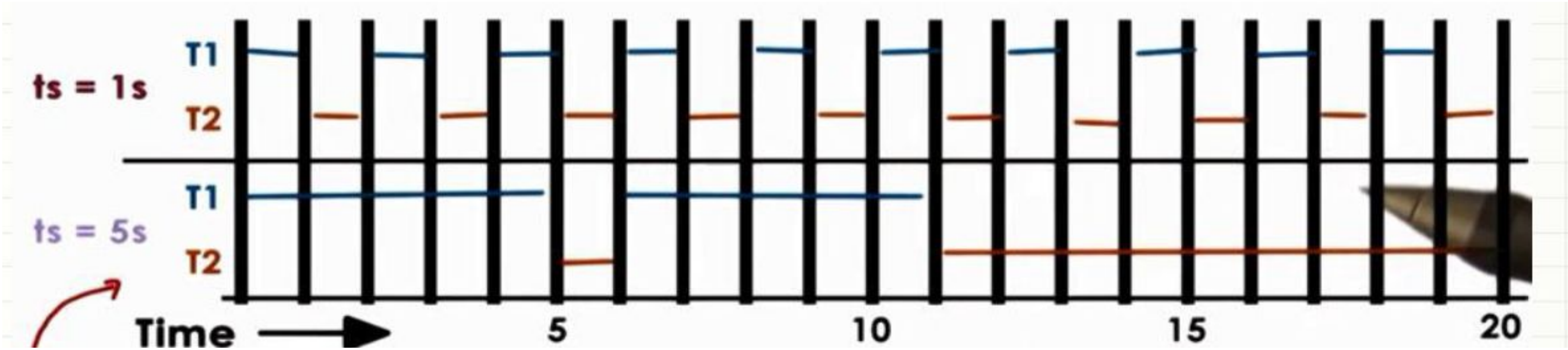
TS	Through put	Avg Wait	Avg comp.
1	0.091	0.55	21.35
5	0.098	2.55	17.75
infinite	0.1	5	15



Time Slicing

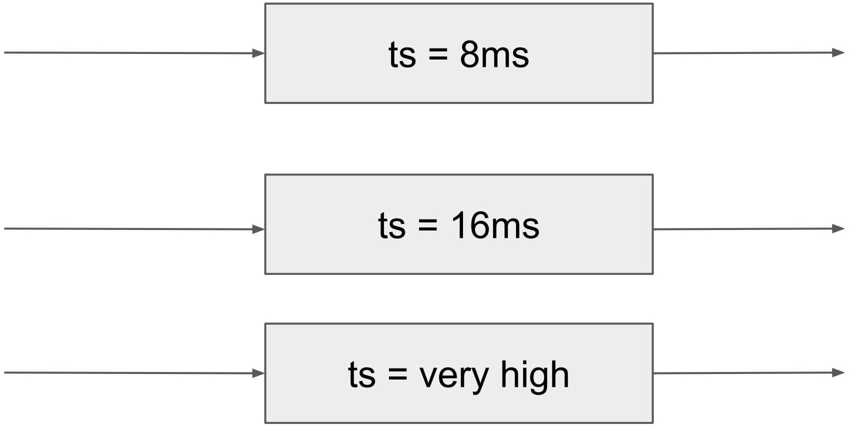
1 CPU Bound task
1 task going for i/o every sec
i/o completes in < 1 sec

TS	Through put	Avg Wait	Avg comp.
1	0.091	0.55	21.35
5	0.091	0.55	21.85
infinite	0.082	2.55	17.75



Observation:

- i/o bound tasks perform better with smaller time slice
- CPU bound prefers larger ts
- Build 1 run queue and based on task type (cpu or i/o bound) assign ts
  OR
- Build multiple run queue



I/O intensive - highest priority

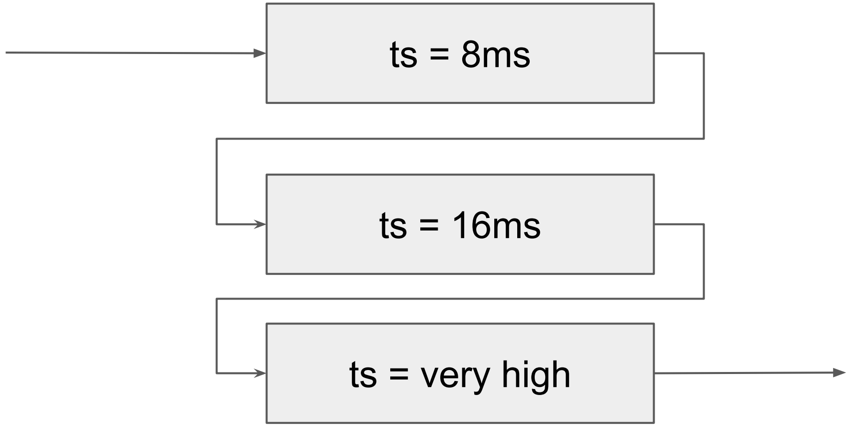
Mix of i/o & CPU

Low priority - CPU intensive

Issue with multiple queues approach:

- How to know if a task is i/o or CPU intensive
- Historical data? How about a new type of task
- How about a task which changes behaviour ?

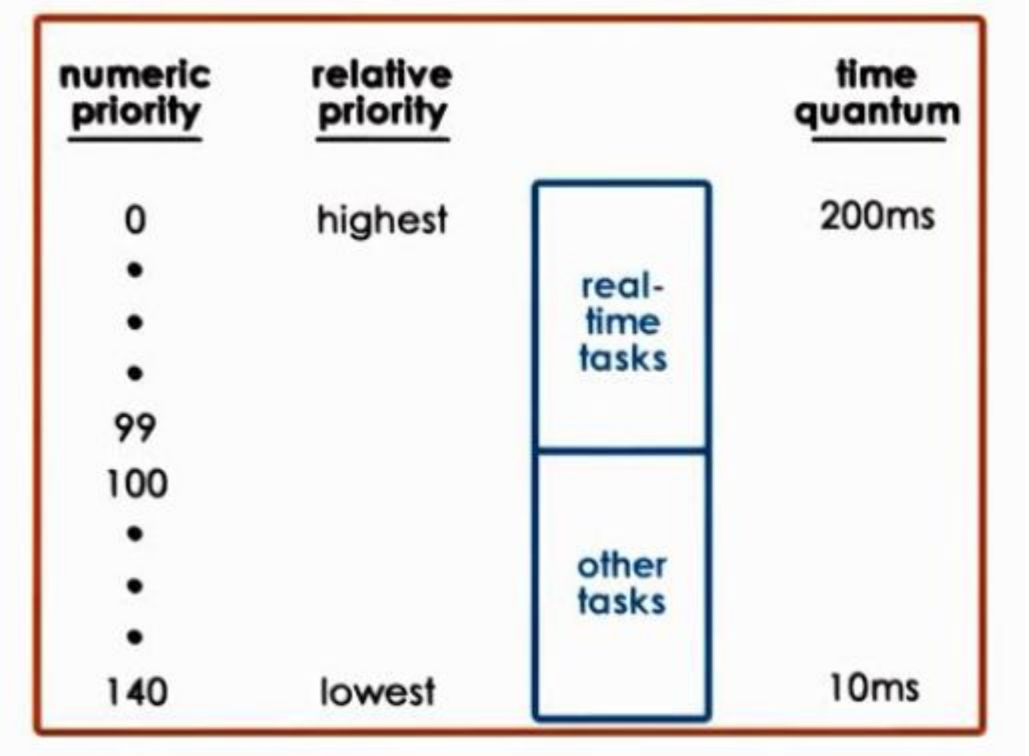
Solution: Multi level feedback queue by Fernando Corbato



- Add task to top queue
- Push down to lower level if ts expires
- Boost priority by pulling up on i/o wait

O(1) Linux (2.5)

- O(1) time to select/add tasks
- Supports preemptive and priority
- Timeslice: depends on priority, smallest for low priority and highest for high
- Feedback: average sleep time
  - Longer : p=p-5
  - Smaller: p=p+5
- 2 queues : active and expired
  - When active is empty use expired
  - Because the arrays are accessed only via pointer, switching them is as fast as swapping two pointers
- Issue:
  - Perf of interactive tasks
  - Not fair

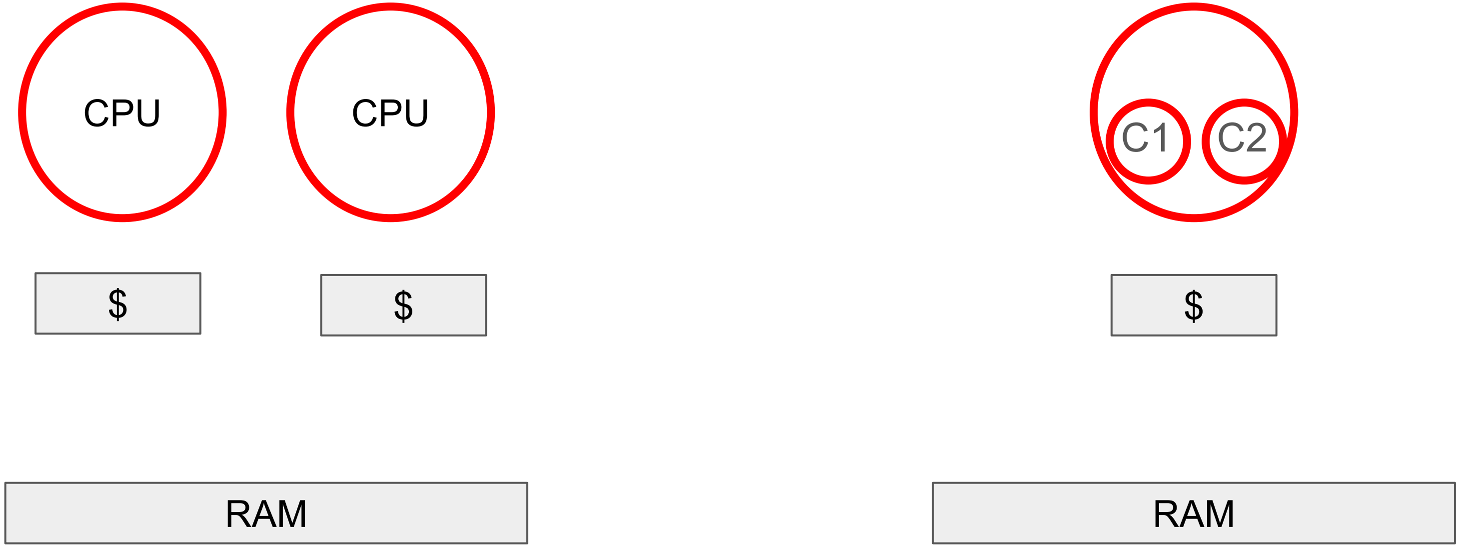


CFS Linux (>2.5)

- Red black tree
- Ordered by virtual run time
- Always pick leftmost - task with smallest v run time
- Continuously monitor tasks - preempt if a task with lesser v run time exists and put back to tree
- Rate faster for low priority, rate slower for higher priority
- Same data structure (RBT) for all tasks
- add/select task = O(log n)
- Interactive tasks are not waited for unpredictable amount of time

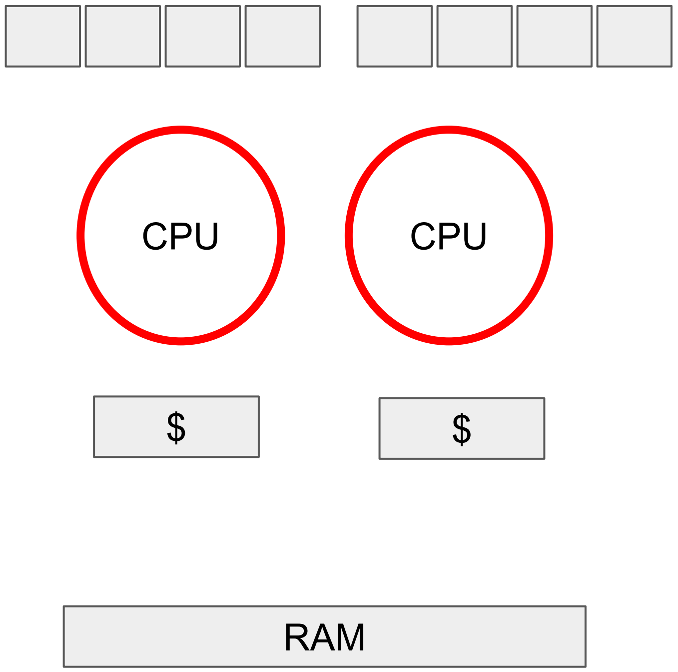
  If the process spends a lot of its time sleeping, then it's spent time value is low and it automatically gets the priority boost when it finally needs it

Multi Core and Multi CPU - Shared memory multi processor (SMT)



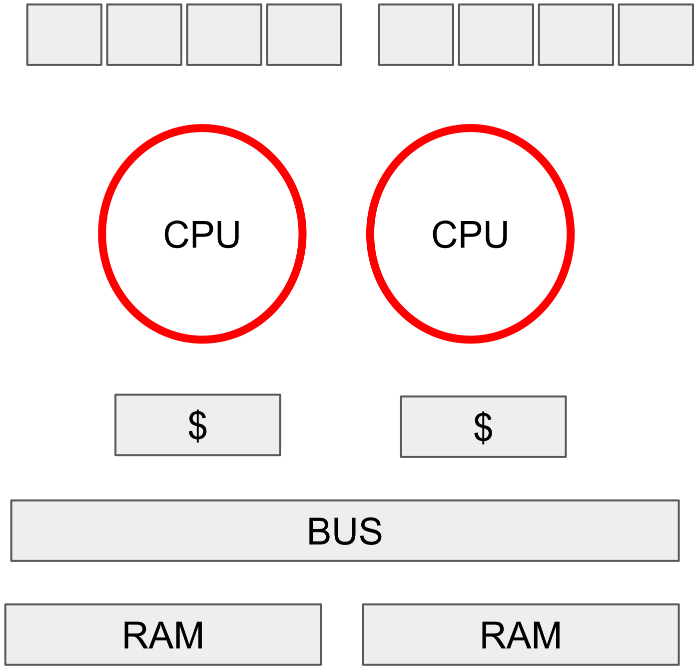
Multi core has private L1, L2 cache while shared last level cache (LLC)

Cache Affinity



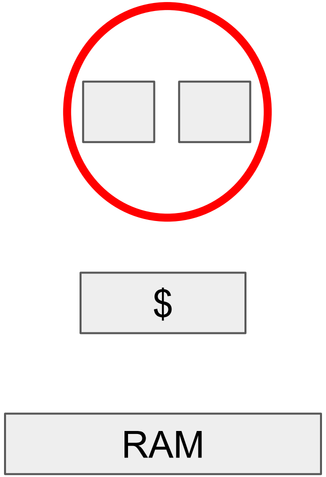
- Keep task on same CPU as possible - higher cache hit to miss ratio
- Use LB
- LB and scheduler per CPU
- Load balance based on queue length or CPU idle time

RAM Affinity

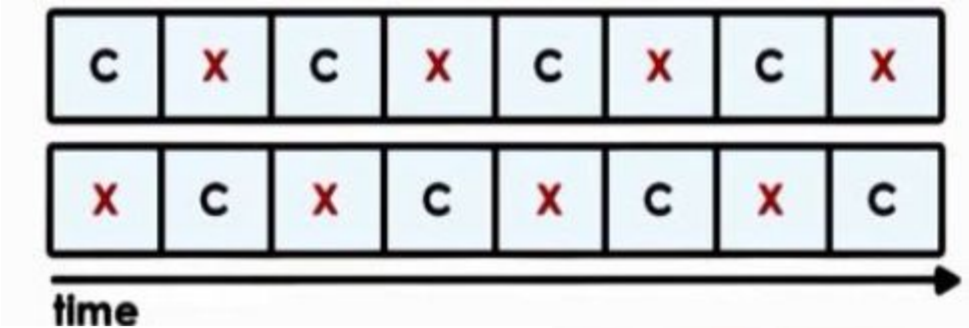


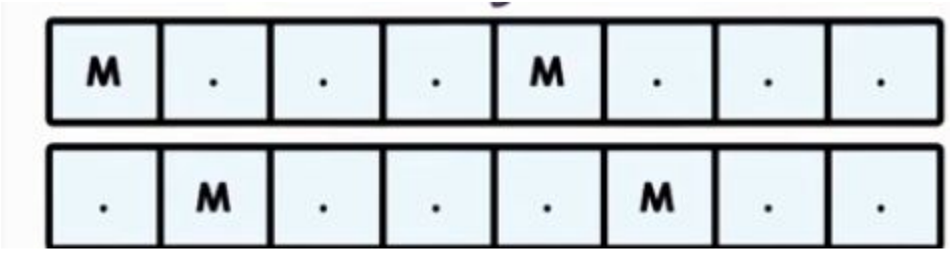
- NUMA - non uniform memory access
- Multiple memory nodes
- Access to local memory is faster
- Keep task to a RAM nearer to CPU
- NUMA aware scheduling
- Load balancing still needed

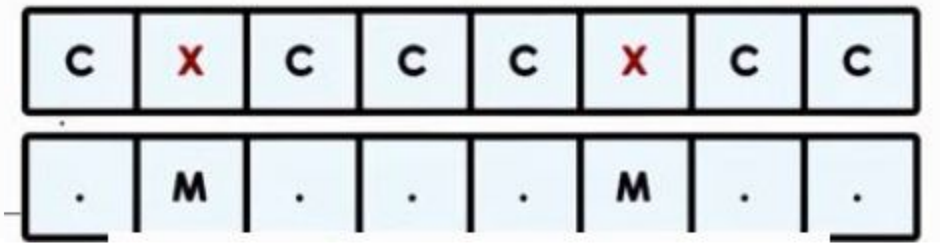
Hyper Threading



- Use Registers to tell execution context
- Single CPU but hardware multithreading (Hyper Threading) can be disabled in bios
- CMT vs SMT (chip and simultaneous)
- Hyperthreading can hide memory latency
- Use If idle time > 2*context switch time
- Use IPC as importance metrics (instructions per cycle)
- Best performance achieved when 1 thread is allocated for compute bound job and other for memory bound







Back to same question - how to know cpu or mem bound task

Use historical data
Use metrics like sleep time - but thread is not sleeping on memory loading
Software takes too long time to compute

Use hardware counters:

- L1, L2, LLC cache misses
- IPC
- Power and energy consumption
- Use linux profile tool (software) which can take decision using hardware counters