Priority Scheduling

Proces	Priority	Burst time
P_1	4	5
P2	2	4
P3	3	6
P4	5	2

Lower the priority no. , ↑ the priority

* When there's a continuous flow of process with shorter lengths, then the longer process will be blocked. This is called starvation. Similarly, a low priority process can be blocked by a stream of high priority processes.

Aging technique :- Gradually increase the priority of a process that wait for long time.

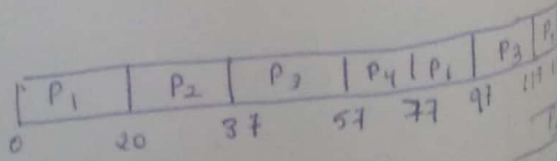
Round Robin Scheduling

Time quantum :- A small unit of time is allotted
(Time slice) to each process.

The ready queue is a circular queue
If the CPU burst is less than time quantum, then
it releases the CPU voluntarily. The CPU is allotted
to next process for 1 time quantum.

TQ = 20

proc	Burst time
P1	53
P2	17
P3	68
P4	24



Waiting time		Turnaround time		Response time	
P_1	81	P_1	134	P_1 0	162 - 68
P2	20	P_2	37	P_2 20	- 94
P3	94	P_3	162	P_3 37	121 - 24
P4	97	P_4	121	P_4 57	
	292		454	114	

Avg = 73 → Avg = 113.5 (Increased compared to SJFS)

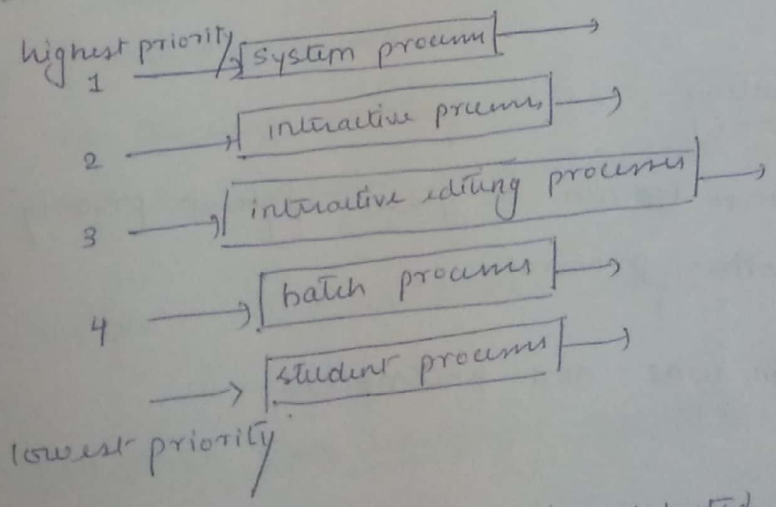
Avg = 28.5 → Best response time

So interactive systems need best response time. Th
∴ Round robin is good for interactive systems.

If time quantum is increased, it degrades to FCFS.
If time quantum is decreased, context switching increases, so the overhead increases.

Time quantum is generally b/w 10 - 100 ms as context switching time < 10 ms.

Multilevel Queue Scheduling



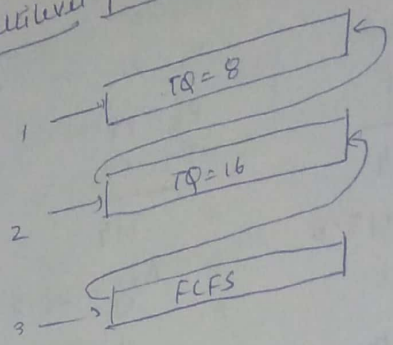
4th queue process will be selected only when 1, 2, 3 are empty. This again results in starvation.

So, time slicing is done

1 → 60%
2 → 20%
3 → 15%
4 → 5%

we can apply diff. scheduling algo. for different queue.

Multilevel feedback queue



Que:

- No. of queue
- Scheduling algo for each queue
- How to upgrade to higher priority queue
- When to demote process to lower priority queue
- Which queue to send a process to when it enters.

enough

Multiprocessor scheduling

Symmetric → any CPU can execute any process from the common ready queue.

Assymetric → multiple queues are maintained.

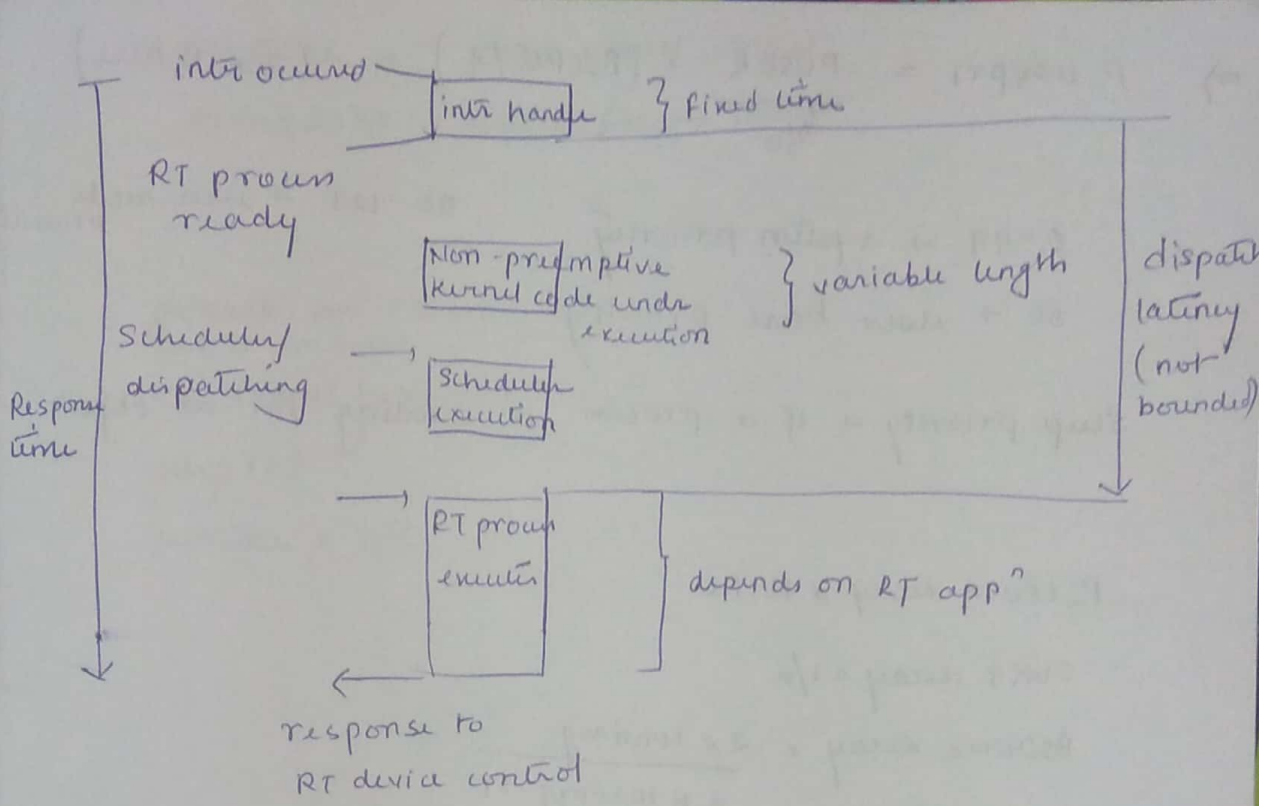
Real time scheduling

→ Real time process ~~has~~ can be given highest priority among all other processes.

→ Old ~~UNIX~~ Windows system was non-preemptive.

old UNIX → user code is fully preemptive
processes inside kernel are non-preemptive

system file UNIX → real time process



Strict bounds are not supported because of unbounded dispatch latency.

Multiple preemption points are added to support/favour RT processes.

Evaluation methods

1) Deterministic modeling
2) Queuing modeling
3) Implementation → most accurate way but costly

→ depends on given dataset only

Traditional UNIX scheduling algorithm :—

variable priority based round robin scheduling algo.

Parameters

P_CPU ← Recent CPU usage

P_NICE ← nice value (user defined priority) 0-39

P_pri ← overall priority (current priority of the proc)

P_usrpri ← user mode priority

⇒ P_usrpri = PUSER + (P_cpu/4) + (2 * P_nice)

50

50-127 → user mode

0-99 → system priority

50 → user base priority

Sleep priority → if a proc is waiting for an object

P_cpu = decay * P_cpu

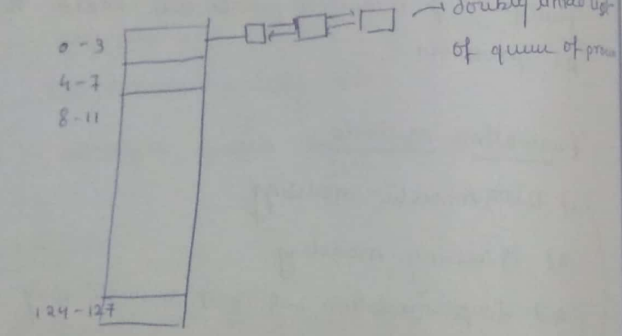
SVR3 decay = 1/2

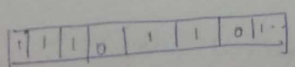
BSDUnix decay = $\frac{2 * loadavg}{2 * loadavg + 1}$

Data-structure

Multilevel priority queue

32-levels





which queue is empty & which is not-

0 → empty

1 → non-empty

Search the bit-array for 1st set 1.

FFC → finds 1st bit queue

INSQHI } insert/remove process from queue.
RMQHI }

SAVE PCTX → save proc context

RSTORPCTX → restore proc context

generate one proc
put it in queue

t = (rand() % 10) + 1;

sleep(t)

runtime + = t

Synchronisation

Full cond^n

```
while ((in+1) % MaxItems == out)
  {

  }
```

```
volatile int in, out, count;
```

Empty cond^n

```
while (in == out)
```

In order to diff. b/w full & empty cond^n count is used.

```
producer(){
    while(1){
        while(count == MaxItems); // do nothing
        buffer[in] = item;
        in = (in+1) % MaxItems;
        count++;
    }
}

consumer(){
    while(1){
        while (count == 0); // do nothing
        item = buffer[out];
        out = (out+1) % MaxItems;
        count--;
    }
}
```

Both may not function correctly when executed concurrently bcoz count++ & count--

```
S1:  MOV R1, count
S2:  INC R1
S3:  MOV count, R1
      :
P1:  MOV R2, count
P2:  DEC R2
P3:  MOV count, R2
      :
```

initially count = 5

S1	R1 = 5
P1	R2 = 5
P2	R2 = 4
P3	count = 4
S2	R1 = 6
S3	count = 6

wrong value.

race condition :- when several process access and manipulate the same data concurrently & the outcome depends on the particular order in which the access takes place.

critical section :- portion of code which is manipulating shared variable

No 2 process should execute critical section at the same time it should be mutually exclusive.

```
do{
    Entry code
    critical section
    Exit code
    remainder section
}while(true);
```

Properties of correct sol^n :-

1) Mutual exclusion :- If procn P_i is executing in its critical section, then no other processes can be executing in their critical sections.

2) Progress property :- If no procn is executing critical sect^n & some processes wish to enter their critical sect^n, then only those procns who are not executing in their remainder sections can participate in deciding which will enter its critical section next & this selection can't be postponed.

3) Bounded waiting :- There should be a limit on the no. of times other processes are allowed to enter their critical sect^n after a procn has made request to enter its critical sect^n. & before that request is granted.

Critical section problem

General solution

2-process critical section

wrong & both.

```
int turn = i ;
while( turn != i ) ;     // do nothing
critical section
turn = j ;
```

When i is in critical section, turn != j so it can't enter into critical section → mutual exclusion.

when j is not interested to enter critical section, even i cannot enter because turn = j. So progress property is not satisfied.

(modification of soln.

```
int flag[2] = {false, false};

flag[i] = true;          // i is interested
while (flag[j] == true); // if j is also interested wait

critical sec^n

flag[i] = false;
```

~~when both and try to enter at the same time~~

mutual exclusion ✓
bounded waiting ✓
progress ✓

if both try to enter simultaneously. flag[i] = true & flag[j] = true. Both will wait for each other - i is giving chance to j, j is giving chance to i. None of them enter critical section forever those will be in running state (while loop)

(soln :- combine flag and turn.

```
int turn;
int flag[2] = {false, false};
{
flag[i] = true;
turn = j;
```

while (turn == j) && (flag[j] == true) ;

↓ give chance to other one

↓ if other one is not interested this will fail.

critical section

flag[i] = false ;

1) mutual exclusion property :- If j is in critical sect^n flag[j] = true. If i tries to enter turn = j. So it goes to while loop. As soon as j comes out, flag[j] = false. So ~~not~~ i enters critical section. satisfied.

2) Progress property :- If j is not interested flag[j] = false. So i will enter any no. of times it wants to.

3) Bounded-waiting property :- Turn will be overwritten when both try to enter the critical section simultaneously.

1st time if j enters the critical sect^n & makes the request again, then flag[j] = true turn = i but while loop will be executed as flag[i] = true; So 2nd chance is given to i.

This works only for 2 processes.

N-process soln (Bakery algorithm)

int number[N] = {0, 0 ... 0};
int choosing[N] = {false, false ... };

Ex. A bakery shop. People are standing in queue. first come first serve basis will be followed acc. to token numbers.

When 2 processes compete for same token, they might end up chosing same no. as the token. In such a case we consider order of arrival of process.

1st priority → token no.
and priority → pid (for processes with same token no.)

int number[N] = {0, 0 ... }
int choosing[N] = {false, ... }

i = 1 to N

choosing[i] = true ;
number[i] = max(number[j], ∀j) + 1 ;

~~choo~~
choosing[i] = false ; // already chosen the token

for j = 1 to n do
while (choosing[j]);
while((number[j] > 0) && ((number[j], j) < (number[i], i))) ;

// check if any other process has lower token or (same token & less pid) than valid token.

critical section

number[i] = 0 ;

In order to create this algo, N must be known in advance. But N may be dynamic. Under this condn n won't work.

General solutions

1) H/w level solutions

int flag = {false / true} // shared variable

step1: set flag to true
if oldvalue == true goto step1

```
label: MOV AX, flag
       MOV flag, 1
       CMP AX, 0
       JNZ label
```

} if both processes are in the loop then both may see value of flag as false

↓ Should be done atomically

boolean testand_set (boolean &flag)
{
  boolean v;
  v = flag;
  flag = true;
  return v;
}

for intel processor bts arg, pos → old value is replicated in carry flag
                     btc
(bit test & clear)

swap (flag, v) // another atomic instr^n

```
while (testand_set(flag) == true) ;
```

critical sec^n

```
flag = false;
```

```
v = true;
while (v == true)
    swap (flag, v);
```

critical sec^n

```
flag = false;
```

Randomly next process is selected, so no bounded waiting

Busy waiting wastes CPU cycle.

OS - solution Semaphore

Semaphore is an integer variable which can be initialized. Apart from this there are 2 atomic operations.

p - wait } atomic oper^n
v - signal

```
semaphore s = 1 ;   // s is always initialized 1 for mutual exclusion

wait(s):
    while (s.value <= 0) ;
    s.value--;

signal(s):
    s.value++;
```

} meaning of wait not correct imp

wait(s);

critical section

signal(s);

pseudocode for semaphore :-

```
struct semaphore {
    int flag;
    int value;
    struct proc * waitq;
};

semaphore s = {1, NULL};

wait(s):                                    CLI
HW soln
    s.value--;
    if (s.value < 0)
    {
        block this process;
        add to s.waitq;
    }                                       STI
HW soln
signal(s):                                  CLI
HW soln
    s.value++;
    if (s.waitq != NULL)
    {
        remove one proc P from waitq;
        wakeup proc P;
    }                                       STI
HW soln
```

No busy waiting

FCFS is followed.

✓ mutual excl.

✓ progress

✓ bounded waiting

data structure manipulation is also critical sect^n.

common data structure semaphore is being manipulated by many procs at the same time. This itself is a critical section.

This critical sect^n ~~is~~ may be solved using hardware sol^n. but again busy waiting comes into picture.

So stop the time interrupts so that no context switch occurs but this will work only for single procnor machine for multiprocnor procnor system H/w sol^n is used.