Process Scheduling

- A process which allows one process to use the CPU while other processes's execution are on hold (in waiting state).
- Whenever the CPU becomes idle, the OS (CPU Scheduler) must select one of the processes in the ready queue to be executed and load them in the memory to be executed

A Process Scheduling Algorithm tries to:

- Maximize
  - CPU utilization - keep CPU as busy as possible
  - Throughput - Number of process that finish per unit time
- Minimize
  - Waiting time
  - Response time
  - Turnaround time

WAITING TIME

The amount of time specific process needs to **wait in the ready queue**.
Waiting time = turnaround - burst time

RESPONSE TIME

The amount of time after which a process **gets the CPU for first time** after entering the ready queue.

TURNAROUND TIME

- The amount of time to execute a specific process.
- Calculation of total time spent waiting **to get into the memory**, **waiting in the queue**, and **executing on the CPU.**
- Turnaround = Completion Time - arrival Time

Dispatcher

The dispatcher gives a process control over the CPU after it has been selected by the CPU Scheduler.

It is involved in

- Switching context

- Switching to user mode
- Jumping to the proper location in the user program and restart that program.

It should be as fast as possible because it is invoked during every process switch.

The time taken by the dispatcher to stop one process and start another is known as **Dispatcher Latency**.

Types of CPU Scheduling

- Non-preemptive Scheduling

Once the CPU has been allocated to a process, the process keeps the CPU until it releases the CPU either by terminating or by switching to a waiting state.

- Preemptive Scheduling

The tasks are usually assigned with priorities. At times, it is necessary to run a certain task that has a higher priority before another task although it is running. Therefore, the running task is interrupted for some time and resumed later when the priority task has finished its execution.

Scheduling Algorithms

- First come first serve (FCFS) Algorithm
- Shortest job first (SJF) Algorithm
- Shortest remaining time
- Priority Scheduling
- Round Robin Scheduling
- Multilevel Queue Scheduling
- Multilevel Feedback Queue Scheduling

Scheduling in UNIX

- Unix uses a time sharing system (**Time Slice** or **time quantum**), after the time slice expires it preempts the process and schedules another one
- Every process has a **priority**.
- The kernel executes the algorithm to schedule a process selecting the highest priority process from in the states "ready to run".
- For ***tie*** breaking, the kernel picks the one that has been "ready to run" for the **longest time**.
- Each process table entry contains a **priority field** for process scheduling.
- The processes getting a lower priority means that they have recently used the CPU.

The scheduler on UNIX system belongs to the general class of OS schedulers known as **round robin with multilevel feedback**, meaning that the kernel allocates the CPU to a process for a time quantum, preempts a process that exceeds the time quantum, and feeds it back into one of several priority queues. A process may need many iterations through the "feedback loop" before it finishes.

Algorithm

```
algorithm schedule_process
input: none
output: none
{
        while(no process picked to execute){
                for (every process on run queue)
                        pick highest priority process that is loaded in memory;
                if (no process eligible to execute)
                        idle the machine;
                        /* interrupt takes machine out of idle state */
        }
        remove chosen process from the run queue;
        switch context to that of chosen process, resume its execution;
}
```

It makes no sense the pick the process that is not loaded in the memory, since it cannot execute until it is swapped in.

Scheduling Parameters

The range of priorities can be partitioned into 2 classes:

- User priorities
- Kernel priorities
  - Low kernel priorities - they wake up on receipt of a signal
  - High kernel priorities - they continue to sleep (non interruptible)

Processes with user-level priorities were preempted on their return from the kernel to user mode, and processes with kernel-level priorities achieved them in the `sleep` algorithm.

User level priorities are below a threshold.
Kernel level priorities are above the threshold.

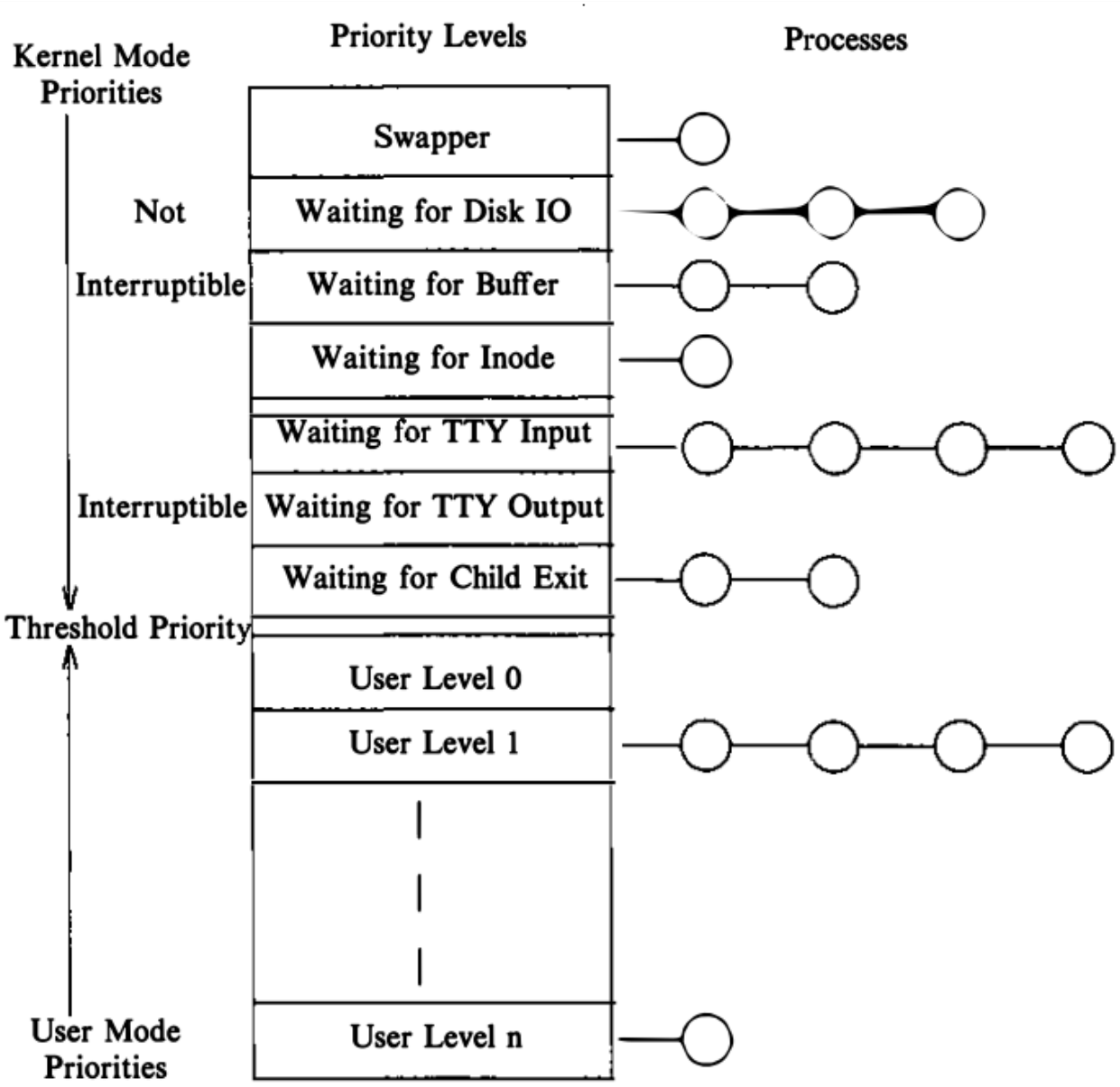


Figure 8.2. Range of Process Priorities

The kernel calculates the priority of a process in specific states:

- It assigns priority to a process about to go to sleep, correlating a fixed, priority value with the reason for sleeping.
  - **Processes that sleep in lower level algorithms tend to cause more system bottlenecks the longer they are inactive.**
  - A process sleeping and waiting for the completion of disk I/O has a higher priority than a process waiting for a free buffer for several reasons, since there is chance that the free buffer may come from the completion of disk I/O.
- The kernel adjusts the priority of a process that returns from the kernel mode to the user mode. The process may entered the sleep state, changing its priority to a kernel level priority. Thus, it may have to lower it when returning to the user mode.

- The clock handler adjusts the priority of all processes in user mode at 1 second intervals to prevent a process from monopolizing the CPU.

Whereas, the kernel does not change the priorities of processes in kernel mode. The kernel does not allow processes with user level priorities to cross the threshold and attain a kernel level priority, unless they make a system call and go to sleep.

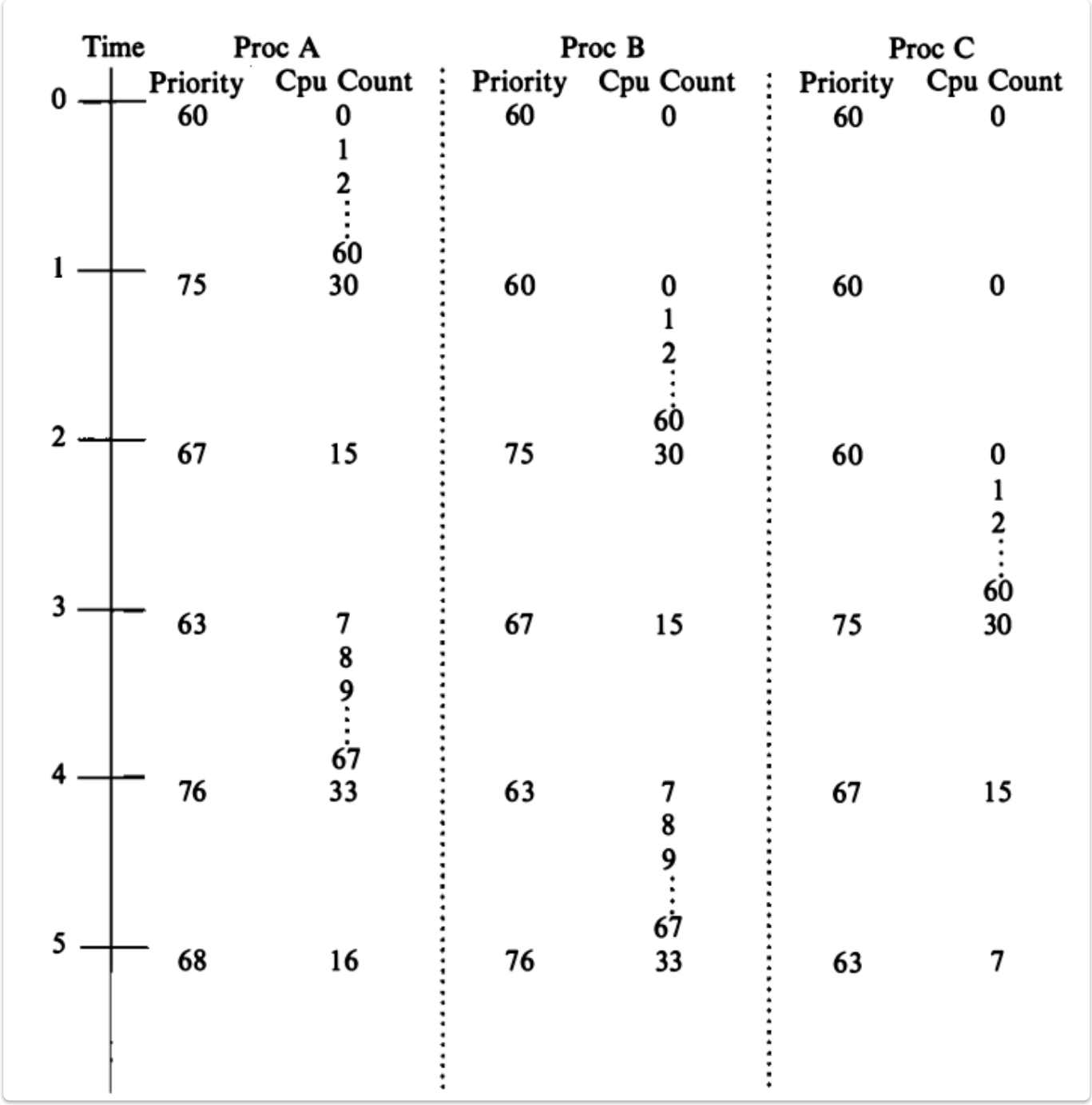
DECAY AND PRIORITY

```
decay(CPU) = CPU/2;
priority = ("recent CPU usage"/2) + (base level user priority (threshold))
```

Numerically low value $\rightarrow$ high priority

Example of Process Scheduling

- Three processes A, B and C with initial priority 60 each.
- The clock interrupts the system 60 times a second.
- They do not make system calls, and no other processes are ready to run.



EXPLANATION

- Assuming A is the first process to run and that it starts running at the beginning of a time quantum, it runs for 1 second.

$$A = 60/2 + 60 = 75$$

- The kernel forces a context switch at the 1 second mark. Process B is scheduled.

$$B = 60/2 + 60 = 75$$

  Meanwhile, A's priority is recalculated

$$A = 15/2 + 60 = 67$$

- Similarly, C is scheduled next since its priority has the numerically lowest value.

$$C = 60/2 + 60 = 75$$

$$B = 15/2 + 60 = 67$$

$$A = 7/2 + 60 = 63$$

- Next A has the highest priority

$$A = 67/4 + 60 = 76$$

$$B = 15/4 + 60 = 63$$

$$C = 60 + 30/4 = 67$$

- Next B has the lowest priority

$$B = 67/4 + 60 = 76$$

$$A = 33/4 + 60 = 68$$

$$C = 60 + 15/4 = 63$$

Process Priorities

Processes can exercise crude control over their priorities using:

```
nice(value);
```

where,

```
priority = ("recent CPU Usage" / 2) + (base priority) + (nice value);
```

Processes inherit the nice value of their parent process during the `fork()` system call. Only the superuser can supply nice values that increase the process priority. Similarly, only the superuser can supply a below a particular threshold.

The `nice()` system call works for the **running process only**; a **process cannot change the priority of another process**.

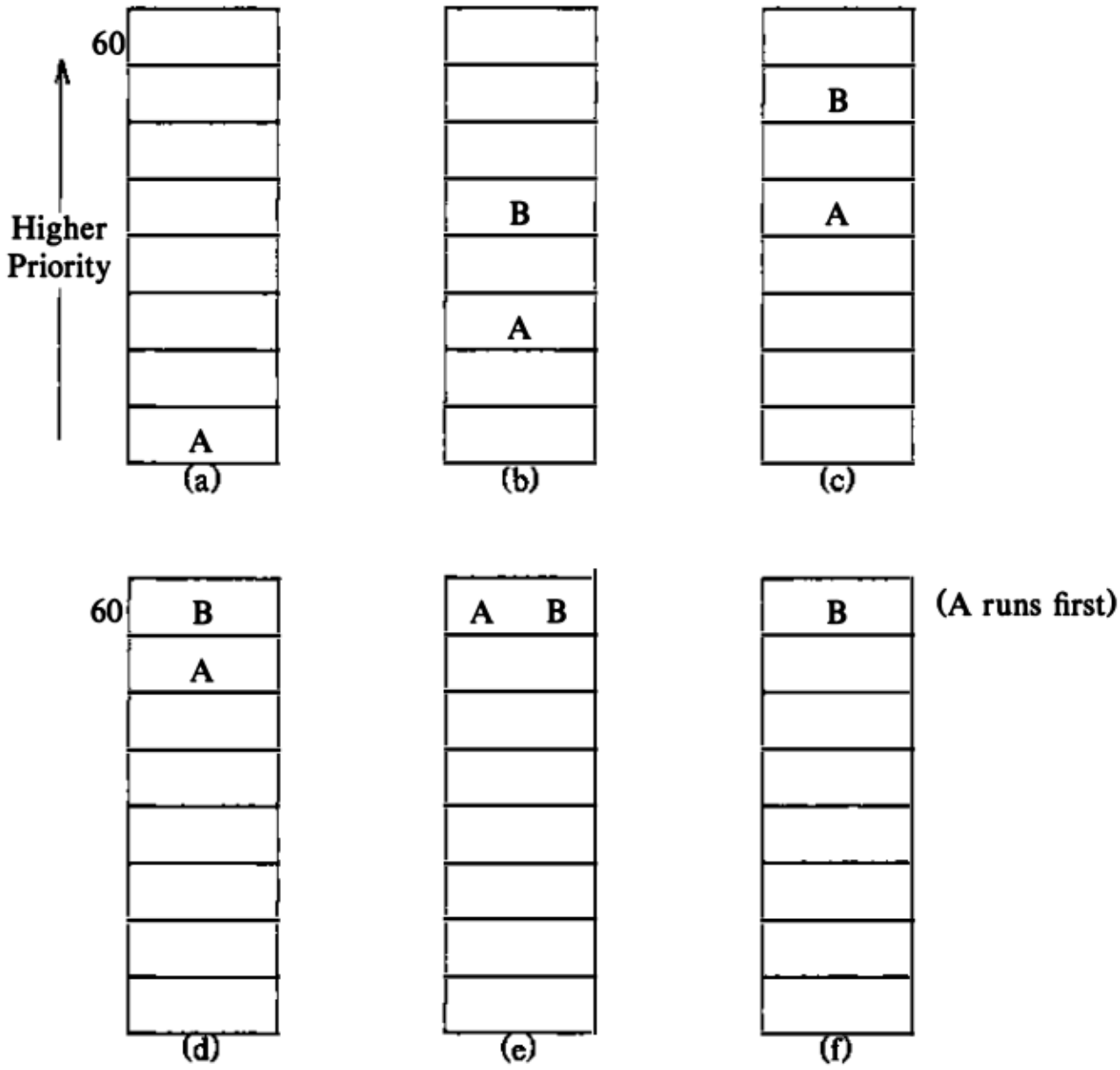


Figure 8.5. Round Robin Scheduling and Process Priorities

- The scheduler does not execute either A or B with B becoming "ready to run" later sometime.
- Over time, their priority can become equal.
- A will run first since it was "ready to run" the longest.

Fair Share Scheduler

- Another field is added to the computation of the priority formula.
- Each process has a new field that points to a fair share CPU usage field, shared by all the processes in the fair share group

EXAMPLE

- Process A is in one group and processes B, C are in another.
- CPU field increments when the process is executed
- Group field is incremented for all processes when a process in the group is executed. (for B and C it will increment together)

$$Priority = base + CPU/2 + Group/2$$

Time	Proc A			Proc B			Proc C		
	Priority	CPU	Group	Priority	CPU	Group	Priority	CPU	Group
0	60	0	0	60	0	0	60	0	0
		1	1						
		2	2						
		⋮	⋮						
		60	60						
1	90	30	30	60	0	0	60	0	0
					1	1			1
					2	2			2
					⋮	⋮			⋮
					60	60			60
2	74	15	15	90	30	30	75	0	30
		16	16						
		17	17						
		⋮	⋮						
		75	75						
3	96	37	37	74	15	15	67	0	15
						16		1	16
						17		2	17
						⋮		⋮	⋮
						75		60	75
4	78	18	18	81	7	37	93	30	37
		19	19						
		20	20						
		⋮	⋮						
		78	78						
5	98	39	39	70	3	18	76	15	18

Real Time Processing

- Implies the capability to provide immediate response to specific external events and, hence, to schedule particular processes to run within a specified time limit after occurence of an event.

- It is desirable that the response be quick.
- A true solution must allow real-time process to exist dynamically, providing them with a mechanism to inform the kernel of their real-time constraints.
- No standard UNIX system has this capability today.