# Time and Scheduling

Outline of material in this set:

- *Time measures.*
  Accounting for CPU time, *e.g.* 50% idle.
- *Performance measures.*
  Measures of CPU performance.
- *Task states.*
  Label indicating a task's needs.
- *Scheduling data.*
  Information OS uses to schedule tasks.
- *Scheduling events.*
  Actions which cause the OS to stop one task and start another.
- *Scheduling algorithms.*
  How the OS chooses which task to run.





# Time Measures

At any time a CPU will be doing one of three things:

- Running a task in user mode,
- running in privileged mode,
- idle (no tasks to run).

For an understood interval, $\mathcal{T}$, let

$t_u(\mathcal{T})$ denote time CPU in **user** mode,

$t_p(\mathcal{T})$ denote time CPU in **privileged** mode,

$t_i(\mathcal{T})$ denote time CPU is **idle**.

The duration of the interval, $t(\mathcal{T})$, is the sum of these ...

$$t(\mathcal{T}) = t_u(\mathcal{T}) + t_s(\mathcal{T}) + t_i(\mathcal{T}).$$

$\mathcal{T}$ sometimes omitted for brevity.





# Performance Measures

Several different measures of performance are used.

Each measures a different aspect of performance.

- *Utilization* [of the CPU].
  How efficiently CPU time is being used.
- *Throughput* [of the system].
  What rate (*e.g.* tasks/hour) work is getting done.
- *Turnaround time* [of a particular or average task].
  The time to complete an individual task ...
  ... or the average time to complete a task.
- *Response time* [of a particular or average task].
  The time between a particular event and response.

  (Usually the task responding to user input.)





## Utilization

The utilization of a CPU over $\mathcal{T}$, denoted $U(\mathcal{T})$ is given by:

$$U(\mathcal{T}) = \frac{t_u(\mathcal{T}) + t_p(\mathcal{T})}{t(\mathcal{T})},$$

where $t_u(\mathcal{T})$ is the user time over interval $\mathcal{T}$,

$t_p(\mathcal{T})$ is the privileged time over interval $\mathcal{T}$,

and $t(\mathcal{T})$ is the total duration of interval $\mathcal{T}$.

Utilization is in the range $[0, 1]$.

Accountants want utilization to be high ...
... users want it to be low (when they run their tasks).

## Throughput

Let $n(\mathcal{T})$ be the number of tasks which complete in time period $\mathcal{T}$.

Then throughput is given by

$$\theta(\mathcal{T}) = \frac{n(\mathcal{T})}{t(\mathcal{T})}.$$

The popular SPECrate benchmarks measure throughput.

## Turnaround Time

Let a task be submitted at $t_1$.

Let the task be completed at $t_2$.

Then the turnaround time for the task is $t_2 - t_1$.

Users want turnaround time to be short.

When utilization is low, turnaround time is usually short.

The SPECint and SPECfp benchmarks measure turnaround time on an unloaded system (in contrast to the SPECrate benchmarks).





## Response Time

Response time defined for an *event* and *response*.

*Event* is something external that task senses.

*Response* is the task's reaction.

To compute response time need ...
... time of event ...
... and time of task's response.

Let event occur at $t_1$ response occur at $t_2$ ...
... then response time is $t_2 - t_1$.

Examples:

Text editor:
*Event*, key pressed; *response*, letter appears on screen.

Pull-down menu:
*Event*, mouse click; *response*, menu appears.

Real-time system:
*Event*, pressure exceeds 500 kPa; *response*, valve opened.

Users want response time to be short.

Zero-cost design choices frequently ...
... improve response time but degrade utilization ...
... or vice versa.

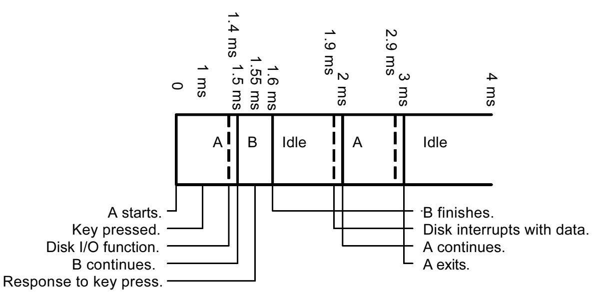
In this course, principally concerned with response time.





## Example Problem

*Find the utilization and throughput for the time interval described below. Find the response time for the event and response described below.*




Event: key pressed at $t = 1\,\text{ms}$.

Response: task B writes a character on the screen at $t = 1.55\,\text{ms}$.

Total time, $t = 4\,\text{ms}$. User time, $t_u = 2.4\,\text{ms}$. Privileged time, $t_p = 0.3\,\text{ms}$. Idle time, $t_i = 1.3\,\text{ms}$

Utilization, $U = (t_u + t_p)/t = 2.7/4 = .675$

Response time, $1.55\,\text{ms} - 1\,\text{ms} = 0.55\,\text{ms}$.

Throughput, $\theta = 2/4\,\text{ms} = 500$ tasks per second.

Throughput and utilization are usually computed for a much larger time interval.





## OS Management of CPU Time

OS determines task to run on CPU by ...

... examining recorded task information ...

... examining *scheduling lists* describing tasks ...

... and applying a *scheduler* to this info to choose task.

### Task Information

Stored in *process control block* (PCB) ...

... a data structure maintained by OS.

OS provides a PCB for each task.

### PCB Includes:

- Current task *state*.
- Resources assigned to task.
- Context information.
- Information for scheduling.

## Task States

Task's state indicates . . .

. . . if it's running . . .

. . . or why it's not running.

The following are a set of possible task states in a simple system:

- **New.**
  Task being created.
- **Ready.**
  Task not running, but could run.
- **Run.**
  Task is running.
- **Wait.**
  Task waiting for something.
- **Zombie.**
  Task finished running, but not yet removed.

### State Assignment

Task initially assigned **New** state.

OS frequently changes task's state.





## Task State Details

### The **New** State

Entered when task created.

Indicates that task incomplete.

Exited after essential resources allocated.

Usual transition from **New** to **Ready**.

### The **Ready** State

Entered from **Run** state when switching to different task.

Entered from **Wait** state when resource becomes available.

Entered from **New** state when task is ready to run.

Exit to **Run** state when OS chooses task to run.

### The **Run** State

Entered from **Ready** state when OS has chosen task to run.

Exit to **Ready** state if OS determines task has had enough time.

Exit to **Wait** state if needed resource not available.

Exit to **Zombie** state at end of execution.





### The **Wait** State

Entered from **Run** state when task needs to wait for some event or resource.

Exit to **Ready** state.

### The **Zombie** State

Entered from **Run** state when task finishes.

In this state the task disappears, so there is no next state.

Number of tasks in **Run** is $\leq$ number of CPUs.

Threads have similar states.





## Scheduling Lists

*Scheduling lists* are lists of tasks.

Each task is in at most one list.

Used by OS for scheduling.

### Reason Task in a List

Tasks in a list are waiting for something. . .
. . .that "something" is determined by the list.

### Typical Scheduling Lists

- *Ready list.*
  Holds all tasks in **Ready** state, waiting for CPU.

  Task to run chosen from ready list.
- *Wait list.*
  Holds all tasks in **Wait** state, waiting for resource or event.

  Wait list checked when resource becomes available. . .
  . . .tasks waiting for resource moved to ready list.

  Wait list similarly checked when event occurs.

### Actual systems use more lists.

*E.g.*, several wait lists might be used, each for different resources.

## Scheduler and Scheduling

*Scheduling:* deciding which, and how long, to run a task.

Which task determined by scheduler.

How long determined by quantum and *preemption* policy.

### Scheduling Procedure

A *scheduling event* occurs . . .

. . . invoking OS (entering kernel) . . .

. . . possibly interrupting a task.

OS uses scheduler to choose new task.

Current task moved from **Run** state.

New task moved to **Run** state.

Timer set to quantum (so OS can regain control).

Context switch and jump to new task.





## The Quantum

OSs designed to divide time between several tasks.

Done by limiting time in **Run** to $\leq$ one *quantum*.

Quantum typically several milliseconds.

Quantum implemented using a timer.

When task put in **Run** state timer set to quantum.

If task runs for duration of quantum. . .

. . .timer will expire, returning control to OS. . .

. . .which will schedule new task.

This will be referred to as *OS preemption* here.
(Another sense of preemption is described below.)

Tasks vary in use of quantum.

Compute-bound tasks frequently use whole quantum.

I/O-bound tasks frequently must wait for I/O before quantum expires.





## Choice of Quantum Length

### Effect of Quantum Length on Efficiency

There is overhead in switching tasks.

The smaller the quantum, the greater the number of task switches.

More context switches means greater overhead.

Therefore, for efficiency, the quantum should be large.

### Effect of Quantum Length on Interactive Users

Interactive users want fast, *e.g.* $< 100\,\mathrm{ms}$, response.

A task in the ready list cannot generate a response.

The smaller the quantum, the less time before a task removed from ready list.

(Task will make more trips to ready list, but each wait will be less.)

Therefore, for interactive users, smaller quantum better.

Real Time users want predictable performance.

A smaller quantum *may* result in more predictable performance.





## Preemption Policy

*Preemption* is the moving of a running task to ready list.

A preempted task could continue to run.

Tasks are preempted to allow other tasks to run.

*Preemption policy* determines when tasks may be preempted.

Using a *task-preemptive* policy preemption occurs anytime.

Otherwise, preemption occurs only when quantum expires.

### Advantages of Task Preemption

Tasks don't wait for lower-priority tasks to finish quantum.

(*E.g.*, when a task moves from **Wait** to **Ready** while lower-priority task running.)

### Terminology Note

Note: 'OS preemptive' and 'task preemptive' are not used outside this class.

Outside this class the term 'preemptive' applies to both systems, the exact meaning must be determined from the context.

In most popular usage, the 'OS-preemptive' sense is intended.

## Preemption and the Kernel

The Problem

Programs operating on shared data ...
... cannot be interrupted at certain times.

An OS kernel is such a program.

Solutions:

Don't allow the kernel to be interrupted.

Easy to implement.

Response times may be too large for RTS.

Used in many conventional operating systems.

Allow the kernel to be interrupted when it's safe ...
... such a kernel is called *preemptable*.

Much more difficult to implement.

Lower maximum response times possible ...
... since lengthy system calls need not block high-priority event.

Used in real-time operating systems.





## Scheduling Events

The OS invokes the scheduler at *scheduling events*.

Scheduler chooses task to run, OS switches tasks.

Scheduling Events Indicate

Current task should be stopped...

...or new task should be started.





## Typical Scheduling Events

*Scheduling events caused by running task:*

- Task requests currently unavailable resource.

  (*E.g.*, a disk read.)

  Task put in wait list, removed after resource available.

- Task "voluntarily" relinquishes CPU.

  (*E.g.*, by executing a *wait* or *sleep* system call.)

  Task put in wait list; removed when "wait" event occurs or at wake-up time.

- Task attempts illegal instruction or memory access.

  (*E.g.*, `int *j=0,i; i=*j;`).

  Task killed.

*Scheduling event planned by OS:*

- Timer expires.

  (*E.g.*, quantum used up.)

  Running task may be replaced by another.





*Other scheduling events:*

- Change in resource status.

  (*E.g.*, disk read completes, memory allocation completes.)

  May cause higher-priority task to become **Ready**...

  ...which scheduler might choose to replace running task.

- Events that need attention.

  (*E.g.*, key pressed, tank pressure exceeds 1 MPa.)

  Events sometimes attended by dæmon tasks...

  ...lurking in wait list (unless attending events).

  Running task put in ready list and...

  ...appropriate dæmon task moved from wait to ready to run.

  (Such events also tended by *interrupt handlers* to be covered later.)

Segue

Due to scheduling event, scheduler called.

What criteria are used to choose task?

## The Scheduler

The *scheduler* chooses a task to run . . .
. . . based on a *scheduling algorithm* . . .
. . . implementing one or more *scheduling policies*.

*Scheduling policy:* simple method of choosing task.

Scheduling algorithm may use multiple policies.

Scheduling algorithm *used* in two ways:

- *On line.*
  Scheduling algorithm used at time of scheduling event.

  Used in conventional and many RT operating systems.

- *Off line.*
  Scheduling algorithm used before system started.

  Result is *schedule* of task run times.

  OS uses schedule to choose task to run.

  This technique used in some RT systems.

Comparison

On-line scheduling bases its choice on prevailing conditions.

Off-line scheduling can guarantee that timing constraints are met.

A system can easily use both techniques.

On-line techniques will be covered first.





## Scheduling: Basic Policies

All policies . . .

. . . start with set of tasks and . . .

. . . return a subset of the tasks.

One task in subset will be chosen to run . . .

. . . perhaps using another policy.

First-Come, First-Served (FCFS) Policy

*Description*

Arrival time *to ready list* recorded for each task.

Ready or running tasks with the smallest arrival time are chosen.

Note: quantum expiration forces running task into ready list . . .

. . . giving it the largest arrival time.

*Example*

Let $a_1 = 1398$, $a_2 = 1140$, and $a_3 = 690$ . . .

. . . be times tasks 1, 2, and 3 entered ready list, respectively.

Then task 3 will leave first, followed by 2, followed by 1.





Priority Policy

Description

Each task is associated with a *priority*.

Priority may be fixed by user . . .

. . . or it may be changed by the OS.

Tasks with the highest priority are chosen.

Priority specified by an integer.

Higher integer will indicate higher priority. (Unlike Unix.)

For example, suppose . . .

$p_7 = 3$, $p_{99} = 5$, and $p_3 = 2$,

. . . are currently in the ready list, . . .

. . . where $p_i = j$ indicates task $i$ has priority $j$.

Then task 99 will be the next chosen, followed by 7, followed by 3.

Priority Changed by OS

The OS might adjust the priority to improve response time.

*E.g.*, task receiving user input . . .
. . . might have its priority temporarily increased.





Round Robin Policy

Description

Tasks are partitioned into *classes*.

Classes are arranged in some circular sequence.

Initially, OS chooses tasks in first class.

OS records which class the running task was chosen from.

Next task chosen from next non-empty class in sequence.

Since sequence is circular, first class in the sequence follows last class in the sequence.

## Round Robin Example

Suppose the following sequence of classes is used:

{undergraduate, graduate, faculty, background, dæmon}.

The 'undergraduate' class contains all tasks started from undergraduate computer accounts, the 'graduate' class contains all tasks started from graduate computer accounts, etc.

Suppose the ready list contains tasks of classes

$c_7$ = graduate, $c_5$ = undergraduate, $c_2$ = faculty, and $c_1$ = undergraduate,

where $c_i$ is the class of task $i$.

Suppose the previous task chosen from the ready list was in the 'undergraduate' class.

Then the task to be chosen must be in the 'graduate' class.

This can only be task 7.

(An additional scheduling policy needed if any class can contain more than one member.)





## Random or Arbitrary Policies

Description

Choose task randomly or choose task with lowest process ID.

Choice of task is <u>not</u> based upon anything related to timing.

Used to break ties ...
... *e.g.*, two tasks with the same priority.





## Nearest-Deadline First Policy (Deadline Scheduling)

Description

Each task has a *deadline*, the time at which it's required to finish.

Tasks with smallest (nearest) deadline is chosen.

Used in RTS.

This is a "best effort" policy: deadlines may not be met.

For example, suppose ready list contains two tasks, 103 and 6.

Deadlines for these tasks are

$t = 6000$ for task 103 and $t = 5200$ for task 6.

Task 6 is chosen first.





## Example Problem

*An operating system uses a priority scheduler with a* 200 ms *quantum and is not task-preemptive. The table below describes the tasks which are in the ready list at* $t = 0$. *(It is known beforehand how much CPU time tasks will use.) None of the tasks have gotten CPU time before* $t = 0$. *Draw a diagram showing the task states and what the CPU is doing from* $t = 0$ *until the last task finishes.*

| Task Name | Priority | Run Time | Other Information |
|---|---|---|---|
| $A$ | 3 | 200 ms | Disk read: 100 ms + 50 ms |
| $B$ | 2 | 500 ms | |
| $C$ | 1 | 100 ms | Disk read: 20 ms + 50 ms |

where "Disk read: $x + y$" means that a disk read will be issued after $x$ CPU time; the disk will take $y$ to return the data.

Solution highlights:

$t/\,\mathrm{ms} \in [0, 100]$: $A$ in **Run** state, $B$ and $C$ **Ready**.

At $t = 100\,\mathrm{ms}$ $A$ issues a disk read, goes to **Wait** state, $B$ goes to **Run** state.

At $t = 150\,\mathrm{ms}$ disk read completes, $A$ goes from **Wait** to **Ready** state, $B$ continues to run.

At $t = 300\,\mathrm{ms}$ $B$'s quantum is used up; $B$ goes from **Run** to **Ready**; $A$ goes from **Ready** to **Run**.

At $t = 400\,\mathrm{ms}$ $A$ finishes execution, $B$ goes from **Ready** to **Run**.

At $t = 600\,\mathrm{ms}$ $B$'s quantum is used up. Since $B$ is the higher priority ready task, it gets another quantum.

At $t = 700\,\mathrm{ms}$ $B$ finishes, $C$ goes from **Ready** to **Run** (finally).

At $t = 720\,\mathrm{ms}$ $C$ issues a disk read, going from **Run** to **Wait**. There are no remaining tasks in the ready list so the CPU idles.

At $t = 770\,\mathrm{ms}$ the read completes, $C$ goes from **Wait** to **Ready** to **Run**.

At $t = 850\,\mathrm{ms}$ $C$ finishes, the CPU goes idle.





## Combining Scheduling Policies

Idea

A policy selects a subset of tasks in ready list.

Scheduler supposed to choose one task.

(Or zero tasks if the ready list is empty.)

Therefore to obtain one task several policies applied.

Rounds

Policies applied in some order.

Each application called a *round*.

First application called round 1, etc.

Example

Round 1: Priority.

Round 2: FCFS.

After round 1, subset may contain several tasks with same priority.

FCFS policy in round 2 used to choose one of these.





Combining Scheduling Policies into Trees

Round 1 is root of tree; a single policy is used.

Let a priority policy be used in round $i$.

Let there be $P$ possible priorities.

Let $p_i$ denote priority of tasks chosen in round $i$.

In round $i+1$ one of $P$ possible policies is used.

Policy used determined by $p_i$.

(The $P$ policies form branches of tree.)

Example

Round 1: Priority policy with 3 possible priorities.

Round 2: Policies: (1) FCFS, (2) FCFS, (3) Deadline.

"Ties" between priority-1 and -2 tasks broken using FCFS.

Ties between priority-3 tasks broken using the deadline policy.

Stopping Tasks

Quantum and task preemption may depend upon position in tree.

*E.g.*, larger quantum for lower priority tasks.





## Example, Combining Scheduling

*An operating system uses the following scheduling algorithm. In the first round, a three-level priority policy is used. In the second round two different policies are used. Those tasks in level three (from the first round) are selected using the nearest-deadline-first policy. The tasks in the other two levels are selected using the FCFS policy. The OS is task-preemptive. Draw a diagram showing task states and CPU activity in the interval* $t \in [0, 1000\,\mathrm{ms}]$ *for the tasks described in the table below. The OS uses a* $100\,\mathrm{ms}$ *quantum.*

| Task Name | Round-1 Priority | Arrival | Run Time | Other Information |
|---|---|---|---|---|
| $A$ | 3 | 300 ms | 50 ms | Deadline at 500 ms |
| $B$ | 3 | 290 ms | 200 ms | Deadline at 550 ms |
| $C$ | 2 | 40 ms | 300 ms | |
| $D$ | 2 | 30 ms | 3 days | Disk read: $\mathrm{mod} 90\,\mathrm{ms} + 70\,\mathrm{ms}$. |
| $E$ | 1 | 0 ms | 1 year | |

where "Disk read: $\mathrm{mod} x + y$" means that a disk read will be issued after every $x$ of CPU time (*e.g.*, $x$, $2x$, $3x \ldots$); the disk will take $y$ to return the data.

Solution (Simulator Output):

Time 0

Task E created.

Task E changing from Ready to Run

Time 30

Task D created.

Task E changing from Run to Ready

Task D changing from Ready to Run

Time 40

Task C created.

Time 120

Task D requests unavailable resources.

Task D changing from Run to Wait

Task C changing from Ready to Run

Time 190

Resources now available for task D.

Task D changing from Wait to Ready





Time 220

Task C quantum expired.

Task C changing from Run to Ready

Task D changing from Ready to Run

Time 290

Task B created.

Task D changing from Run to Ready

Task B changing from Ready to Run

Time 300

Task A created.

Task B changing from Run to Ready

Task A changing from Ready to Run

Time 350

Task A finishes normally.

Task A changing from Run to Zombie

Task B changing from Ready to Run





Time 450

Task B quantum expired.

Task B changing from Run to Ready

Task B changing from Ready to Run

Time 540

Task B finishes normally.

Task B changing from Run to Zombie

Task C changing from Ready to Run





Time 640

Task C quantum expired.

Task C changing from Run to Ready

Task D changing from Ready to Run

Time 660

Task D requests unavailable resources.

Task D changing from Run to Wait

Task C changing from Ready to Run

Time 730

Resources now available for task D.

Task D changing from Wait to Ready

Time 760

Task C finishes normally.

Task C changing from Run to Zombie

Task D changing from Ready to Run

Time 850

Task D requests unavailable resources.

Task D changing from Run to Wait

Task E changing from Ready to Run

14-37

Time 920

Resources now available for task D.

Task D changing from Wait to Ready

Task E changing from Run to Ready

Task D changing from Ready to Run