# 5.1 SCHEDULING ALGORITHMS

## 5.1.1 Basic Concepts

First of all, some basic concepts must be introduced and explained. Scheduling is central to operating system design. The success of CPU scheduling depends on two executions. The first one is the process execution consisting of a cycle of CPU execution and Input/ Output (I/O) wait. The second one is the process execution, which begins with a CPU processing, followed by I/O processing, then followed by another CPU processing, then another I/O processing, and so on. The CPU I/O Processing Cycle is the basic concept of processor technology. The processing time is the actual time that is required to complete some job.

The CPU scheduler selects from among the processes in memory that are ready to execute, and allocates the CPU to one of them. CPU scheduling decisions take place when a process is switching from run- ning to waiting state; switching from running to ready state; switching from waiting to ready and terminating.

Beside the CPU scheduler, dispatcher is also a basic and important concept in processor technology. The dispatcher module gives control of the CPU to the process selected by the short-term scheduler, and this involves: switching context, switching to user mode, and jumping to the proper location in the user program to restart that program. Most dispatchers have dispatch latency, which is the time they take for the dispatcher to stop one process and start another running.

Then we discuss some criteria of scheduling.

CPU Utilization. The CPU utilization refers to a computer's usage of processing resources, or the amount of work handled by a CPU, and it is used to gauge system performance. Actual CPU utilization varies depending on the amount and type of

managed computing tasks. The first aim of processor technology is increasing the CPU utilization by keeping the CPU as busy as possible.

**Throughput**. The throughput means the amount of processes that complete their execution per time unit.

**Turnaround Time.** The turnaround time means the amount of time to execute a particular process, and it can be calculated as the sum of the time waiting to get into memory, waiting in the ready queue, and executing on the CPU and the I/O.

**Waiting Time**. The waiting time means the amount of time a process has been waiting in the ready queue.

**Response Time**. The response time means the amount of time it takes from when a request was submitted until the first response is produced.

**Completion Time**. The completion time of one job means the amount of time needed to complete it, if it is never preempted, inter- rupted, or terminated.

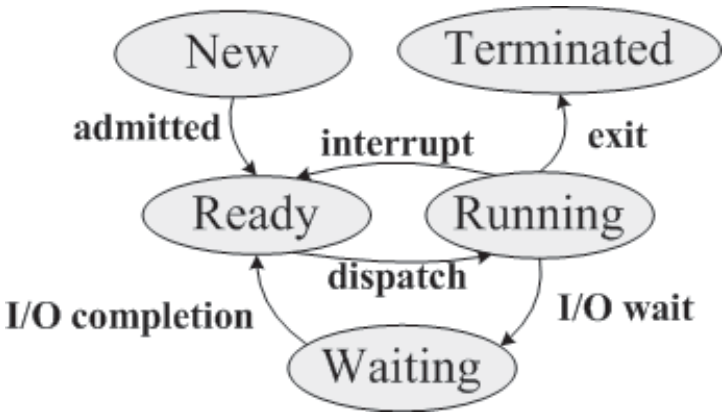



Figure 5.1 The diagram of the process states.

As shown in Fig. 5.1, processes have five types of states. At the *new* state, the process is in the stage of being created. At the *ready* state, the process has all the resources available that it needs to run, but the CPU is not currently working on this process's instructions. At the *running* state, the CPU is working on this process's

instructions. At the *waiting* state, the process cannot run at the moment, because it is waiting for some resource to become available or for some event to occur. At the *terminate* state, the process was completed.

### 5.1.2 First Come, First Served Scheduling Algorithm

An important measurable indicator of processor is the average com- pletion time of jobs. Fig. 5.2 represents an example of the schedule for k jobs. As shown in the figure, there are $k$ jobs, marked as $j_k$, to be completed in the processor. The first job $j1$ requires $t1$ time units so that the job $j1$ can be finished by time $t1$. The second job $j2$ starts after the fist job $j1$ is finished, and the required length of time is $t2$. Therefore, the second job $j2$ can be accomplished by the time $t1 + t2$. Repeat this procedure until the last job $j_k$ is done.

The total completion time:

$$A = t1 + (t1 + t2) + (t1 + t2 + t3) + ... + (t1 + t2 + t3 + ... + t_k)$$

$$= k * t1 + (k - 1) * t2 + (k - 2) * t3 + ... + t_k$$

(5.1)


Figure 5.2 A schedule for k jobs.

One of the simplest scheduling algorithm is First Come, First Served (FCFS). The

FCFS policy is widely used in daily life. For ex- ample, it is the standard policy for the processing of most queues, in which people wait for a service that was not prearranged or preplanned. In the processor technology field, it means the *jobs* are handled in the orders.

For instance, there are four jobs, *j*1,*j*2,*j*3, and *j*4, with different processing times, which are 7, 4, 3, and 6 respectively. These jobs arrive in the order: *j*1,*j*2,*j*3,*j*4. In FCFS policy, they are handled by the order of *j*1,*j*2,*j*3,*j*4, as shown in Fig. 5.3. The waiting time for j1 is 0, for *j*2 is 7, for *j*3 is 11, and for *j*4 is 14. The average waiting time is (0+7+11+14)/4 = 8. The average completion time is [7 + (7+4) + (7+4+3) + (7+4+3+6)] / 4 = 13.

Suppose that the jobs arrive in the order *j*2,*j*3,*j*4,*j*1; the result produced by using FCFS is shown in Fig. 5.4. The waiting time for *j*1 is 13, for *j*2 is 0, for *j*3 is 4, and for *j*4 is 7. The average waiting time is (13+0+4+7)/4 = 6. The average completion time is [4 + (4+3) + (4+3+6) + (4+3+6+7)] / 4 = 11. Both the average waiting time and the average completion time of this scheduling is less than the previous one.

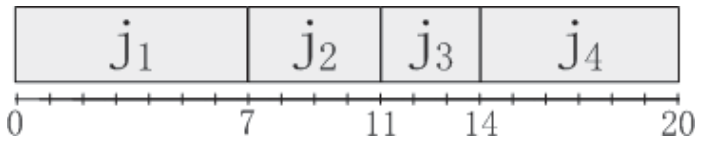



Figure 5.3 An example of FCFS scheduling.

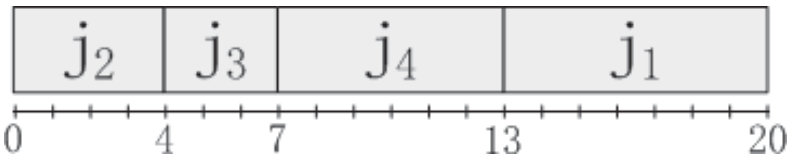



Figure 5.4 Another FCFS result if changing arrival sequence.

### 5.1.3 Shorted Job First Scheduling Algorithm

Then we will introduce another scheduling policy, which is Shortest Job First (SJF). SJF is a scheduling policy that selects the waiting process with the smallest execution time to execute first. SJF is advantageous because of its simplicity, and it minimizes the average completion time. Each process has to wait until its execution is complete.

Using the example mentioned in Section 2.2, while ignoring their arrival time, we first sort these jobs by their processing time, as $j3$, $j2$, $j4$, $j1$. The SJF scheduling result is shown in Fig. 5.5. The waiting time for $j1$ is 13, $j2$ is 3, $j3$ is 0, and $j4$ is 7. The average waiting time is (13+3+0+7) = 5.75. The completion time for $j1$ is (13+7), $j2$ is (3+4), $j3$ is (0+3), $j4$ is (7+6). The average completion time is (20+7+3+13)/4 = 10.75. This scheduling has lower average waiting time and average completion time than the previous two schedules.

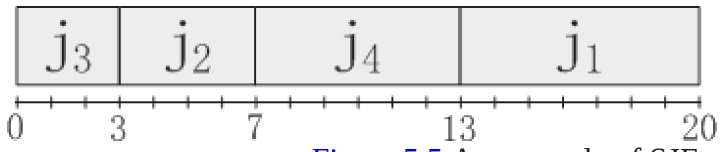



Figure 5.5 An example of SJF scheduling.

Theorem: *SJF scheduling has the lowest total completion time with a single processor.*

Proof by contradiction: Assuming that there are a series of jobs that were sorted by their completion time from short to long, as $j_1, j_2, j_3, \ldots, j_i, j_{i+1}, \ldots, j_k$, which also means the completion time of them can be ordered as $t_1 < t_2 < t_3 < \cdots < t_i < t_{i+1} < \cdots < t_k$. Using the SJF scheduling algorithm, the result is exactly the same as the or- der $j_1, j_2, j_3, \ldots, j_i, j_{i+1}, \ldots, j_k$. Then we suppose that there is another order A that has lower total completion time than the one produced by SJF, $j_1, j_2, j_3, \ldots, j_{i+1}, j_i, \ldots, j_k$. Based on Equation 5.1, the total completions time is $T = k * t_1 + (k-1) * t_2 + (k-2) * t_3 + \cdots + (k-i+1) * t_i + (k-i) * t_{i+1} + \cdots + t_k$. So, we can get the total completion time of both

orders. The SJF one is $T_s = k * t_1 + (k-1) * t_2 + (k-2) * t_3 + \cdots + (k-i+1) * t_i + (k-i) * t_{i+1} + \cdots + t_k$. The A one is $T_a = k*t_1+(k-1)*t_2+(k-2)*t_3+\cdots+(k-i+1)*t_{i+1}+(k-i)*t_i+\cdots+t_k$. From the supposing condition, $T_s < T_a$.

$T_s > T_a$;

$k*t_1+(k-1)*t_2+(k-2)*t_3+\cdots+(k-i+1)*t_i+(k-i)*t_{i+1}+\cdots+t_k$

$>k*t_1+(k-1)*t_2+(k-2)*t_3+\cdots+(k-i+1)*t_{i+1}+(k-i)*t_i+\cdots+t_k$; $(k-i+1) * t_i + (k-i) * t_{i+1} > (k-i+1) * t_{i+1} + (k-i) * t_i$;

$t_i > t_{i+1}$.

However, $t_i > t_{i+1}$ is contradictory to $t_i < t_{i+1}$, in the assum- ing condition. As a result, A does not exist, which means there is no solution that has lower total completion time than the SJF scheduling. In the end, we can conclude that SJF scheduling has the lowest average waiting time with a single processor. However, is SJF still optimal with multiple processors?

### 5.1.4 Multiprocessors

After discussing the single processor, we will expand the topic into multiprocessors. There are nine jobs with different completion times in three processors, as shown in Fig. 5.6, and we first give an opti- mal schedule using SJF. The average completion time is {(3+5+6) + [(6+10)+(5+11)+(3+14)] + [(3+14+15)+(5+11+18)+(6+10+20)] }
/ 9 = 18.33. There is another optimal schedule, as shown in Fig. 5.7.
The average completion time is {(3+5+6) + [(5+10)+(3+11)+(6+14)]
+ [(5+10+15)+(6+14+18)+(3+11+20)] } / 9 = 18.33.

In multiprocessors, there are three theorems:

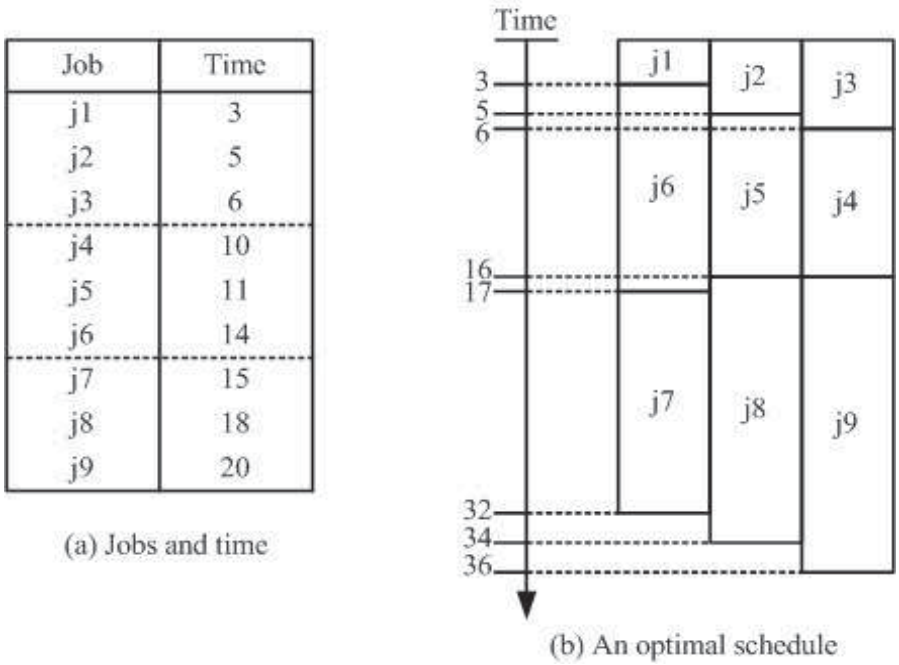



Figure 5.6 An SJF schedule to complete nine jobs in three processors.

Theorem 5.1 *SJF scheduling has the optimal average waiting time and completion time in the multiprocessor.*

Theorem 5.2 *With the same average waiting time, there is more than one schedule with various final completion time.*

Theorem 5.3 *The algorithm to find the optimal final completion time is NP-Hard.*

Assuming that the processing time of $j1$ to $j3_k$ is $t1$ to $t3_k$, respec- tively, the average completion time in three processors calculates as Equation 5.2: The average completion time is

$$\{(t1 + t2 + t3) + (t1 + t2 + t3 + t4 + t5 + t6) + \cdots + (t1 + t2 + \cdots + t3_k)\}/3k$$

$$= \{k(t1 + t2 + t3) + (k-1)(t4 + t5 + t6) + \cdots + (t3_{k-2} + t3_{k-1} + t3_k)\}/3k. \quad (5.2)$$

Then we assign that $T1 = t1 + t2 + t3$, $T2 = t4 + t5 + t6, \ldots,$

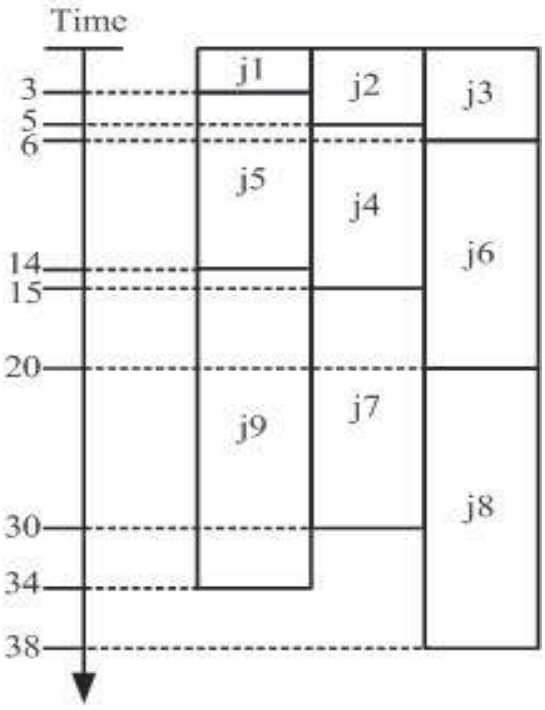



Figure 5.7 Another schedule to complete nine jobs in three processors.

$T_k = t3_{k-2} + t3_{k-1} + t3_k$. The total completion time in three processors can be formulated as $kT1 + (k-1)T2 + \cdots + T_k$. At last, we can formulate this problem into the one in a single processor. In the end, we can use

the same method as the one in Section 2.3 to prove that the SJF schedule has the optimal average completion time in multiprocessors. From Equation 5.2, we can see that the detailed sequence of $j1, j2, j3$ does not impact the average waiting time of the whole schedule. As a result, the two schedules in Fig. 5.6 and Fig. 5.7 have the same average waiting time. However the time when the last job is completed these two schedules are different, which are 36 and 38. If there is a time constraint that is less than 38, the second schedule is not suitable, while the first schedule can be chosen. Furthermore, there are many other schedules having the same average waiting time with these two schedules, because changing the sequence of $j_{3i+1}, j_{3i+2}, j_{3i}$ does not change the average waiting time. Nevertheless, the time when the last job is completed

is various, and how to find the optimal schedule that has the least time when the last job is completed is too hard to be solved by normal algorithms. This problem is a typical NP-Hard problem, and we will discuss this problem and how to solve it in later chapters.

### 5.1.5 Priority Scheduling Algorithm

The next scheduling algorithm is Priority Scheduling algorithm. In priority scheduling, a priority number, which can be an integer, is as- sociated with each process. The CPU is allocated to the job with the highest priority, and the smallest integer represents the highest prior- ity. The priority scheduling can be used in the preemptive and nonpre- emptive schemes. The SJF scheduling is a priority scheduling, where priority is the predicted next CPU processing time. The following is a given example about the implementation of the priority scheduling in preemptive schemes, as shown in Fig. 5.8. The priority of each job is in- verse with its processing time. As a result, the result using the priority scheduling algorithm is the same as the result from SJF scheduling.

The priority scheduling has the potential restrictions deriving from process starvations. The *Process Starvation* is the processes that re- quire a long completion time, while processes requiring shorter com- pletion times are continuously added. A scheme of "*Aging*" is used to solve this problem. As time progresses, the priority of the process in- creases. Another disadvantage is that the total execution time of a job must be known before the execution. While it is not possible to exactly predict the execution time, a few methods can be used to estimate the execution time for a job, such as a weighted average of previous exe- cution times.

At last, we will introduce the Round Robin (RR) scheduling. In RR scheduling, each job gets a small unit of CPU time, called *time quantum*, usually 10 - 100 milliseconds.

After this time has elapsed, the job is preempted and added to the end of the ready queue. If there are *n* jobs in the ready queue and the *time quantum* is *q*, then each job gets 1/*n* of the CPU time in chunks of at most *q* time units at once. No job waits more than ($n - 1$) time units. If the *q* is large, the RR scheduling will be the FCFS scheduling. Nevertheless, if the *q* is small, the overhead may be too high because of the too-often context switch.

| Jobs | BurstTime | Priority | Arrival Time |
|---|---|---|---|
| $j_1$ | 7 | 4 | 0.0 |
| $j_2$ | 4 | 2 | 2.0 |
| $j_3$ | 3 | 1 | 4.0 |
| $j_4$ | 6 | 3 | 5.0 |

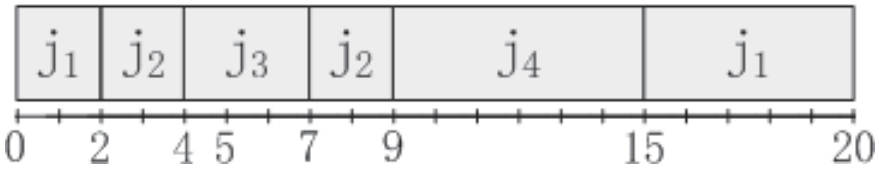



Figure 5.8 An example of the priority scheduling.

Actually, there are two kinds of scheduling schemes that are non- preemptive and preemptive.

***Nonpreemptive.***

The nonpreemptive scheduling means that once the CPU has been allocated to a process, the process keeps the CPU resource until it releases the CPU either by terminating or switching to a waiting state.

***Preemptive***.

In the preemptive schemes, a new job can preempt CPU re- sources, if its CPU

processing length is less than the remaining time of the current executing job. This scheme is known as the Shortest-Remaining-Time-First (SRTF).

In computer science, preemption is the act of temporarily interrupting a job being carried out by a computer. It is normally carried out by a privileged job on the system that allows interruptions. Fig. 5.5 shows SJF scheduling in the situation when all the jobs arrive at the same time, but situation will be complicated when considering their different arrival times, especially in preemptive scheme.

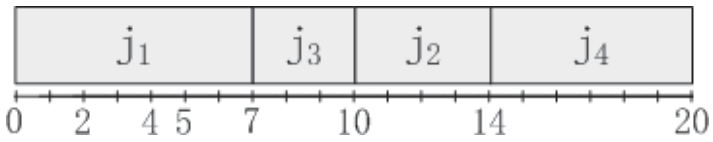



Figure 5.9 An example of the nonpreemptive SJF solution.

Still taking the example mentioned in Section 5.2.2, add arrival times to them, $j1$ arriving at time 0.0; $j2$ arriving at time 2.0; $j3$ ar- riving at time 4.0; $j4$ arriving at time 5.0. The SJF scheduling in a nonpreemptive scheme is shown in Fig. 5.9. At time 0, $j1$ arrives, and there are no other jobs competing with it, so $j1$ is in the running list. At time 2, 4, and 5, $j2$, $j3$, and $j4$ arrive, respectively.

However, they cannot interrupt $j1$ and grab the resource $j1$ is using, so they are all in the waiting list. At time 7, $j1$ is finished, and now there are three jobs in the waiting list. Among these three jobs, $j3$ needs the shortest processing time, so it gets the resource and turns into the running list. At time 10, $j3$ is finished, and now there are two jobs in the waiting list, which are $j2$ and $j4$. Since $j2$ needs a shorter processing time than $j4$ does, $j2$ gets the resource and turns into the running list. At time 14, $j2$ is finished, and now there is only one job in the waiting list, which is $j4$. So $j4$ gets the resource and turns into the running list. Finally, $j4$ is finished at time 20. In this scheduling, the waiting time for $j1$ is 0, $j2$ is (10-2), $j3$ is (7-4),

and $j4$ is (14-5). The average waiting time is (0+8+3+9)/4 = 5. The completion time for $j1$ is 7, $j2$ is (14-2), $j3$ is (10-4), and $j4$ is (20-5). The average completion time is (7+12+6+15)/4 = 10.

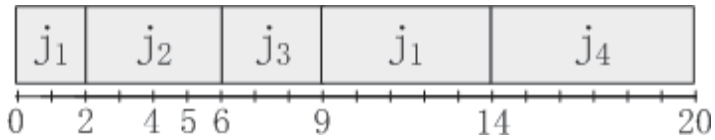



Figure 5.10 Example of the preemptive SJF solution.

The SJF scheduling in a preemptive scheme is shown in Fig. 5.10. At time 0, $j1$ arrives, and there are no other jobs competing with it, so $j1$ is in the running list. At time 2, $j2$ arrives, and $j2$ has shorter processing time than $j1$, so it preempts $j1$. $j1$ goes to the waiting list, while $j2$ in the running list. At time 4, $j3$ arrives. $j3$ needs 3 time to be completed, while $j2$ needs 2 time. So $j3$ cannot preempt $j2$ and stays in the waiting list. At current stage, $j1$ and $j3$ are both in the waiting list.

Next, at time 5, $j4$ arrives, but it has longer processing time than $j2$, so it cannot preempts $j2$. $j4$ joins in the waiting list. At time 6, $j2$ is finished, and now there are three jobs in the waiting list. Among them, $j3$ needs the shortest processing time, so $j3$ get the resource, while others are still waiting. At time 9, $j3$ is finished, and now there are two jobs in the waiting list. Since $j1$ needs a shorter processing time, which is 5, than $j4$ does, which is 6. $j1$ gets the resource and turns into the running list. At time 14, $j1$ is finished, and now there is only one job in the waiting list, which is $j4$. As a result, $j4$ get the resource and is finally finished at time 20. In this scheduling, the waiting time for $j1$ is 9-2, $j2$ is (0), $j3$ is (6-4), and $j4$ is (14-5). he average waiting time is (7+0+2+9)/4 =4.5. The completion time for $j1$ is 14, $j2$ is (6-2), $j3$ is (9-6), and $j4$ is (20-5). The average completion time is (14+4+3+15)/4

= 9.

### 5.1.6 ASAP and ALAP Scheduling Algorithm

First, we will introduce the Directed Acyclic Graphs (DAG) to model the scheduling problem about the delay in processors. A DAG is a directed graph with no directed cycles. It is formed by a collection of vertices and directed edges, each edge connecting one vertex to another. There is no way to start at some vertex and follow a sequence of edges that eventually loop back to this vertex. We create a DAG with a source node and a sink node, as shown in Fig. 5.11. The source node is $V0$, and the sink node is $V_n$. The solid lines refer to the execution delay between nodes. Broken lines mean there is no execution delay between nodes. For example, neither source node nor sink node has the execution time.

Moreover, students need to understand two concepts before intro- ducing the algorithm, including *Predecessor* and *Successor*. A *Prede- cessor* refers to the node that needs to be finished before the current node. For example, in Fig. 5.11, $v2$ and $v3$ are the predecessors of $v5$.

A *Successor* refers to the node that succeeds the current node. In Fig. 5.11, $v4$ is $v1$'s successor.

As exhibited in Fig. 5.11, we define $V = \{v0, v1 , \ldots v_n\}$ in which $v0$ and $v_n$ are pseudo nodes denoting the source node and sink node, respectively. $D = \{d0, d1, \ldots , d_n\}$ where $d_i$ denotes the execution delay of $v_i$;

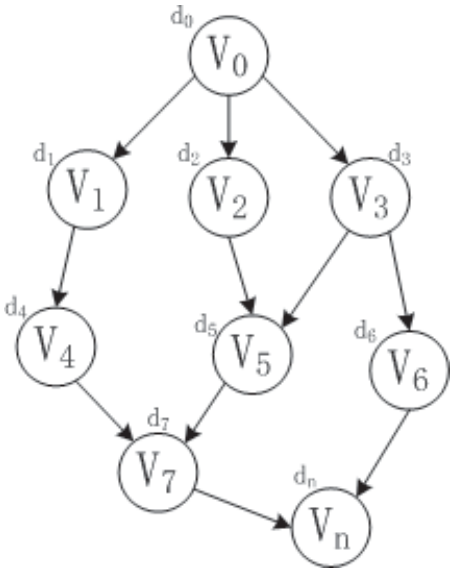
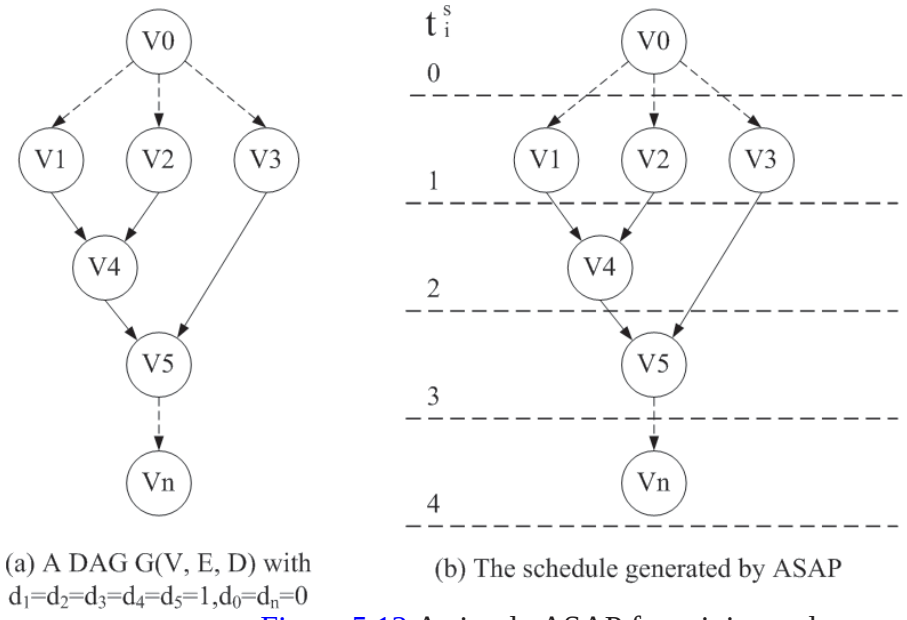



Figure 5.11 A sample of the directed acyclic graph.

Then we use a topological sorting algorithm to produce a legal sequence, which is scheduling for uniprocessor. A topological sorting of a directed acyclic graph is a linear ordering of its vertices, such that for every directed edge $\{u, v\}$ from vertex $u$ to vertex $v$, $u$ comes before $v$ in the ordering. First, finding a list of nodes whose indegree = 0, which means they have no incoming edges, inserting them into a set $S$, and removing them from $V$. Then starting the loop that keeps removing the nodes without incoming edges until $V$ is empty. The output is the result of topological sorting and the scheduling for the uniprocessor. Referring to Fig. 5.11, we can get three results: {$v0$, $v1$, $v4$, $v7$, $v_n$}, { $v0$, $v2$, $v5$, $v7$, $v_n$ }, and { $v0$, $v3$, $v6$, $v_n$ }.

To eliminate the latency, we assign values to $d_i$ and simplify the problem. We set $d1$, $d2$, $d3$, $d4$, and $d5$ as 1. We use two scheduling algorithms, which are As-Soon-As-Possible (ASAP) and As-Late-As- Possible (ALAP) Scheduling Algorithms.

#### *5.1.6.1 ASAP*




(a) A DAG G(V, E, D) with $d_1=d_2=d_3=d_4=d_5=1, d_0=d_n=0$

(b) The schedule generated by ASAP

Figure 5.12 A simple ASAP for minimum latency scheduling.

As shown in Fig. 5.12, first, set $t_0^s = 1$, and $v0$ has no predecessors, and $d0$ is 0. Thus, $v0$ has the same latency as its successors, $v1$, $v2$ , and $v3$. In this step, $v0$ is scheduled. Then because $v1$'s predecessor $v0$ is scheduled, it can be selected at the 1 latency time. The same operations can be implemented with $v2$ and $v3$ at the first latency time unit. In this step, $v1$, $v2$, and $v3$ are scheduled. Then $v4$ can be selected at the 2 latency, because its predecessors, $v1$ and $v2$, are scheduled. However, $v5$ cannot be selected at the 2 latency, because one of its predecessors, $v4$, is not scheduled before the 2 latency. Then after $v4$ is scheduled, $v5$ can be selected at the 3 latency, because its predecessors, $v3$ and $v4$, are scheduled. At last, $v_n$ is selected at 4 latency, because its predecessor, $v5$ is scheduled.

In ASAP for minimum latency scheduling algorithm:

Step 1: schedule $v_0$ by setting $t_0^s = 1$. This step is for launching the calculation of the algorithm.

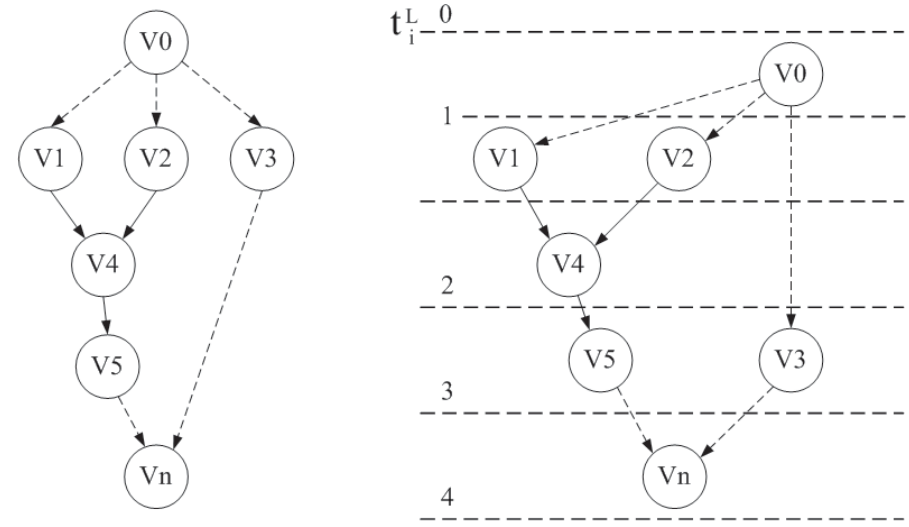
Step 2: select a node $v_i$ whose predecessors are all scheduled. This process will be

repeated until the sink node $V_n$ is selected.

Step 3: schedule $v_i$ by setting $t^s = \max_{j:v_j \to v_i \in E} t^s + d_j$. The equation represents the current node status at the exact timing unit. It represents the latency time at the current node is summing up the maximum latency time of the predecessors' nodes.

Step 4: repeat Step 2 until $v_n$ is scheduled.

#### *5.1.6.2 ALAP*




(a) A DAG G(V, E, D) with $d_1=d_2=d_3=d_4=d_5=1, d_0=d_n=0$

(b) The schedule generated by ALAP when latency-constraint 3

Figure 5.13 ALAP scheduling for latency-constraint scheduling.

As shown in Fig. 5.13, first, schedule the button node $v_n$ at the time latency 3+1, and set $t^L_n = 4$. In this step, $v_n$ is scheduled. Then $v_3$ and $v_5$ can be selected at the 3 time latency, because their successor, $v_n$, is scheduled. In this step, $v_3$ and $v_5$ are scheduled. Then $v_4$ can be selected at the 2 time latency, because its successor, $v_5$, is

scheduled. In this step, $v_4$ is scheduled. In this time latency, although $v_0$ is the predecessor of $v_3$, it cannot be selected at the 2 time latency, because $v_0$'s other successors, $v_1$ and $v_2$, are not scheduled. Then $v_1$ and $v_2$ can be selected at the 1 time latency, because their successor, $v_4$, is scheduled. In this step, $v_1$ and $v_2$ are scheduled. At last, $v_0$ can be selected at 1 time latency, because its successor, $v_1$, $v_2$, and $v_3$, are scheduled, and $d_0$ is 0.

In ALAP for latency-constraint ($\lambda$) scheduling algorithm:

Step 1: schedule $v_n$ by setting $t^L = \lambda + 1$. This step means the first scheduled node is $v_n$.

Step 2: select a node $v_i$ whose successors are all scheduled. It means the selected node must be a node whose successors must be scheduled. This process will be repeated until the source node $v_0$ is selected.

Step 3: schedule vi by setting tL = min tL+dj. The equation represents the current node status at the exact timing unit. It represents that the latency time at the current node is subtracting the sum of minimum latency times from the sink node's latency- constraint.

$$i \quad j : v_j \rightarrow v_i \in E \quad j$$

Step 4: repeat Step 2 until $v_0$ is scheduled. Fig. 5.13 exhibits an ALAP scheduling for latency-constraint scheduling.

Comparing ASAP and ALAP scheduling as shown in Fig. 5.12 and Fig. 5.13, we can find that $v_3$ can be completed at several time latencies. It can be completed at 1 time latency as soon as possible, and 3 time latency as late as possible.

In this section, we introduce some basic concepts, such as CPU utilization, waiting time, response time, and completion time. Then we introduce some scheduling

algorithms, including *First-Come, First Server*, *Shortest-Job-First*, *priority scheduling*, *Round Robin*, *As-Soon- As-Possible*, and *As-Late-As-Possible*. In the next section, we introduce the processor technology about scheduling algorithm in single processor and multi-processor.