CHAPTER 2: CONCURRENCY AND PARALLELISM

In this chapter our goal is to explicitly calculate bounds on the mean throughput rate of executing a single transaction type or a given transaction workload mix for a particular class of models of computer communication systems. In many instances, this provides a *fundamental* limitation on system performance: if a packet switch can only switch one hundred packets per second, then no amount of chicanery (e.g., clever scheduling, increasing the degree of multiprogramming, adopting a new paging strategy, and so forth) will allow this packet switch to switch two hundred packets per second. On the other hand, we must ask ourselves what is the *best* that we might do: is there any intrinsic reason why the packet switch can switch at most one hundred packets per second, or is it possible to increase this to five hundred packets per second by judicious changes in hardware, operating system kernel, data base manager, and application code?

The crux of performance analysis is describing *how* a computer communication system processes each step of a job. The models described here require as inputs a detailed description of the step by step processing of a job, and give as outputs bounds on system performance. By dealing with examples or tractable models of *parallelism* and *concurrency* for computer systems, we hope to build up *intuition* about the benefits of different approaches. A disclaimer is in order: systems that improve performance via added concurrency are in many ways more sensitive to the workload than systems that improve performance via raw speed, because in order to take advantage of concurrency some knowledge of the workload must be used. What happens when the workload changes? Caveat emptor!

2.1 General Approaches

Figure 2.1 shows the two fundamental approaches to parallelism in computer communication systems.

Figure 2.1.Pipeline and Parallel Configurations

The hardware configuration and operating system are fixed: a group of processors interconnected by a high speed bus, with a network operating system to coordinate resource allocation. How do we structure a set of application programs to take advantage of this type of system?

At one extreme of parallelism, the group of processors operates in parallel (i.e., fed by a single queue or work stream) Each processor does all the work for each job. As more processors are added, more work can be done. The total delay will be approximately the processing time of a job, because if we have sufficiently many processors it is highly likely that one will always be available to handle a job.

At the other extreme of parallelism, processors are functionally dedicated to doing a given step of a job. Each step of a job is done sequentially, or in tandem, like a bucket brigade or pipeline of processors. All the input is handled at one stage and all the output at another stage. The pipeline has a great deal of interaction between adjacent stages, unlike the purely parallel case, and hence the potential benefits may not be nearly as great with the pipeline as with the parallel processor case, even though both systems attempt to take advantage of concurrency in the workload.

Finally, we might have an arbitrary network of processors that jobs migrate amongst, which would be a combination of the purely parallel and purely pipeline cases. We will discuss in more detail the two simple cases of parallel and pipeline processing because we can gain insight into the more complicated but realistic situation of the general network.

2.1.1 An Arithmetic Logic Unit An arithmetic logic unit must perform three jobs

Figure 2.2.Arithmetic Logic Unit Block Diagram

[1] Fetch an instruction from memory

[2] Decode the instruction

[3] Execute the instruction

This is an example of a pipeline with three distinct functions.

2.1.2 Floating Point Accelerator A floating point accelerator is a special purpose processor that can accelerate or assist general purpose processors to perform floating point operations at high speed. Here the steps that must be carried out are

Figure 2.3.Floating Point Accelerator Block Diagram

[1] Subtraction of the exponents

[2] Shifting right the fraction from the number with the smaller exponent by an amount corresponding to the difference in exponents

[3] Addition of the other fractions to the shifted one

[4] Counting the number of leading zeroes in the sum

[5] Shifting left the sum by the number of leading zeroes and adjusting the exponent accordingly

Although this partitioning is typical, there are variations in practice to account for overflow and underflow detection, representation of zero, and other number bases.

2.1.3 Additional Reading

[1] P.M.Kogge, **The Architecture of Pipelined Computers,** Hemisphere Publishing, Washington, DC, 1981

2.2 An Example: Printing Ten Jobs

Ten jobs are present at an initial time, say zero. Each job requires $L_K, K=1,...,10$ lines to be printed. These number are given in the table below:

Table 2.1.Lines for Each Job

Job	*Lines*	*Job*	*Lines*
A	L_A=2,000	F	L_F=1,000
B	L_B=1,000	G	L_G=1,000
C	L_C=1,000	H	L_H=2,000
D	L_D=1,000	I	L_I=2,000
E	L_E=2,000	J	L_J=1,000

We see six jobs need one thousand lines to be printed, and four jobs need two thousand lines to be printed. The mean number of lines that must be printed is:

$$L_{average} = \frac{1}{10}\sum_{K=1}^{10} L_K = 1{,}400 \ lines$$

The average is a useful statistic if most jobs require roughly the average (same) number of lines to be printed. That *is* the case here; in fact, six jobs require less than the average, while four require more than the average.

We want to compare the performance of the following configurations

- A single printer that can print one thousand lines per minute
- Two printers that are fed from a common buffer, each of which can print one thousand lines per minute
- A single printer that can print two thousand lines per minute

We will measure total system performance via *mean throughput rate* which is defined as the total number of jobs (denoted by N=10 here) divided by the total time to execute these jobs, denoted by T_{finish}, called the *make span:*

$$mean\ \ throughput\ \ rate = \frac{N}{T_{finish}}\ \ jobs/minute$$

We will measure job oriented performance by the mean time a job, say job K, spends in the system, either waiting to be printed or being printed, until it is completed. We will call this time interval for job K its *flow time* and denote it by $F_K, K=1,...,N=10$. The total mean flow time is the flow time averaged over all jobs:

$$mean\ \ flow\ \ time = \frac{1}{N}\sum_{K=1}^{N} F_K = \frac{1}{10}\sum_{K=1}^{10} F_K \equiv E(F)$$

2.2.1 A Single Printer The single printer case gives us a baseline or benchmark for measuring performance against. It is a natural starting point for virtually any investigation. No matter what order is used to execute jobs, the total number of lines printed for this set of jobs is:

$$L_{finish,single\ \ printer} = \sum_{K=1}^{10} L_K = 14{,}000 \ lines$$

The mean throughput rate is simply the total number of jobs divided by the mean time to print all the jobs:

$$mean\ \ throughput\ \ rate\ \ for\ one\ \ printer = \frac{N}{T_{finish}} = \frac{1}{T_{average}} \quad T_{average} = \frac{T_{finish}}{N}$$

If a single printer can print one thousand lines per minute, then

$$mean\ throughput\ rate = 10\ jobs\ in\ 14\ minutes$$

while if a single printer can print two thousand lines per minute, then

$$mean\ throughput\ rate = 10\ jobs\ in\ (14/2)=7\ minutes$$

Suppose that we print the job stream according to a priority schedule. How will this impact the flow time or time in system statistics? Let's try two special cases to gain insight. The first priority schedule operates as follows: job K is said to have higher priority than job J if $L_K \leq L_J; K, J=1,...,N=10$. In other words, the fewer the lines, the shorter the printing time of a job, the higher its priority. The motivation is to let short jobs get printed quickly, so that they will not have to wait for long jobs, which take a long time anyway. Shortest processing time first scheduling attempts to minimize *job* delay or flow time statistics.

Two additional rules will be used:

- Once a job begins printing, it runs to completion, and cannot be preempted by any other job
- No job can be printed in parallel with itself. For one printer this is no problem, but for two printers this would allow us to split a job, and we do not allow this

We will refer to this schedule as *SPT/NP* (shortest processing time has highest priority, with no preemption) in what follows. The schedule for this case is shown below:

Figure 2.4.SPT/NP Single Slow Printer Schedule

The mean flow time for this schedule for a single one thousand line per minute printer is given by

$$E(F_{SPT/NP}) = \frac{1}{10}[1+2+3+4+5+6+8+10+12+14] = 6.5\ \ minutes$$

For a single two thousand line per minute printer, the mean flow time is simply half this.

The second priority schedule operates as follows: job K is said to have higher priority than job J if $L_K \geq L_J; K, J=1,...,N=10$. In other words, the longer the printing time of a job, the higher its priority. We will refer to this schedule as *LPT/NP* (longest processing time has highest priority, with no preemption) in what follows. The idea for choosing this schedule is that long jobs will take a long time to print, so we might as well begin printing them as soon as possible in order to finish *all* the work as soon as possible. The longest processing time first rule is oriented toward optimizing a *system* performance measure, the fraction of time the system is busy doing work. The schedule for this case is shown below:

Figure 2.5.LPT/NP Single Slow Printer Schedule

The mean flow time for this schedule for a single one thousand line per minute printer is given by

$$E(F_{LPT/NP}) = \frac{1}{10}[2+4+6+8+9+10+11+12+13+14] = 8.9\ minutes$$

For a single printer capable of two thousand lines per minute, the mean flow time is half this.

Each schedule results in the same total system mean throughput rate, but radically different mean flow time statistics.

2.2.2 Two Parallel Printers For the second case, two parallel printers, the worst we could do would be to never use one processor. One alternative is to schedule the jobs, with the shortest execution time jobs having the highest priority:

Figure 2.6.SPT/NP Two Slow Printer Schedule

This means the shortest amount of time to execute all ten jobs is T_{finish}=7.

The mean throughput rate is given by

$$mean\ throughput\ rate\ for\ two\ parallel\ printers_{SPT/NP} = \frac{10}{7} = \frac{1}{0.7\ minute}$$

The speedup compared with a single processor is given by comparing the time to finish all jobs on one processor versus the time to finish all jobs on two processors:

$$speedup_{SPT/NP} = \frac{T_{finish}(P=1)}{T_{finish}(P=2)} = \frac{14}{7} = 2$$

The mean flow time is given by

$$E(F_{SPT/NP}) = \frac{1}{10}[1+1+2+2+3+3+5+5+7+7] = 3.6\ minutes$$

On the other hand, we might print jobs with the longest processing time jobs having the highest priority:

Figure 2.7.LPT/NP Two Slow Printer Schedule

The mean throughput rate for printing jobs is given by

$$mean\ throughput\ rate\ for\ two\ parallel\ printers_{LPT/NP} = \frac{10}{7} = \frac{1}{0.7\ minutes}$$

The speedup over a single printer is given by

$$speedup_{LPT/NP} = \frac{T_{finish}(P=1)}{T_{finish}(P=2)} = \frac{14}{7} = 2$$

Note that the mean throughput rate and the speedup depend upon the scheduling algorithm. The best possible speedup would be to do all the jobs in half the time of a single processor, while the worst would be to just use one processor (and ignore the other):

$$1 \leq speedup \leq 2$$

Finally, the mean flow time per job for this schedule is given by

$$E(F_{LPT/NP}) = \frac{1}{10}[2+2+4+4+5+5+6+6+7+7] = 4.8 \ minutes$$

The mean flow time for scheduling longest jobs first is radically larger than the mean flow time for scheduling shortest jobs first.

2.2.3 Summary We summarize all these findings in the table below:

Table 2.2.Performance Measure Summary

Configuration	*Schedule*	*Mean Throughput Rate*	*Mean Flow Time*
One Slow Printer	SPT/NP	1 job every 1.4 min	6.5 minutes
One Slow Printer	LPT/NP	1 job every 1.4 min	8.9 minutes
Two Slow Printers	SPT/NP	1 job every 0.7 min	3.6 minutes
Two Slow Printers	LPT/NP	1 job every 0.7 min	4.8 minutes
One Fast Printer	SPT/NP	1 job every 0.7 min	3.25 minutes
One Fast Printer	LPT/NP	1 job every 0.7 min	4.45 minutes

What conclusions do we draw?

- Using one fast vs two slow printers has no impact on mean throughput rate
- One fast printer offers slightly better mean flow time compared with two slow printers
- Shortest processing time scheduling results in a lower mean flow time compared with longest processing time scheduling

We will see these lessons repeated later.

2.2.4 Sensitivity One of the primary reasons for carrying out performance analysis studies is to determine the *sensitivity* of the conclusions to parameters. We might not know the number of lines that must be printed for each job, and are only using guesses or estimates. Two types of studies can be done:

- Changing all the numbers by a small amount. For example, we might change the number of lines printed for each job up or down by ten lines or less, and see what changes in mean throughput rate and mean flow time.
- Changing a small set of numbers by a large amount. For example, we might change job J from one thousand lines to *ten* thousand lines.

Let's pursue the second type of study, changing job J from one to ten thousand lines of printing, and study the consequences.

First, what about a single slow printer? The SPT/NP schedule is shown below:

Figure 2.8.SPT/NP Schedule for One Slow Printer (10,000 Lines for Job J)

The *make span* is the time required to completely execute all the work. The make span for this schedule is twenty three minutes, and hence the mean throughput rate is

$$mean\ \ throughput\ \ rate\ =\ \frac{N}{T_{finish,SPT/NP}}\ =\ 1\ \ job\ \ every\ \ 2.3\ \ minutes$$

The mean flow time for SPT/NP is

$$E\left(F_{SPT/NP}\right)\ =\ \frac{1}{10}\ [1+2+3+4+5+7+9+11+13+23]\ =\ 7.8\ \ minutes$$

For a single high speed printer, the mean flow time is simply half of this, and the mean throughput rate is twice as high.

On the other hand, for LPT/NP scheduling on a single slow printer, we see the make span is identical with SPT/NP, as shown in the figure below:

Figure 2.9.LPT/NP Schedule for One Slow Printer (10,000 Lines for Job J)

By inspection from this figure, the mean flow time is

$$E\left(F_{LPT/NP}\right)\ =\ \frac{1}{10}\ [10+12+14+16+18+19+20+21+22+23]\ =\ 17.5\ \ minutes$$

For a single high speed printer, the flow time is half of this, while the mean throughput rate is twice as big.

Next, for two parallel printers, the SPT/NP schedule is shown below:

Figure 2.10.SPT/NP Schedule for Two Slow Printers (10,000 Lines for Job J)

The mean throughput rate is simply

$$mean\ \ throughput\ \ rate\ =\ \frac{N}{T_{finish,LPT/NP}}\ =\ \frac{10}{16\ \ minutes}$$

while the mean flow time is

$$E\left(F_{SPT/NP}\right)\ =\ \frac{1}{10}[1+1+2+2+3+4+5+6+7+16]\ =\ 4.7\ \ minutes$$

Finally, for two parallel printers, the LPT/NP schedule is shown below:

The mean throughput rate is simply

$$mean\ \ throughput\ \ rate_{LPT/NP}\ =\ \frac{N}{T_{finish,LPT/NP}}\ =\ \frac{10}{12\ \ minutes}$$

while the mean flow time is

$$E\left(F_{LPT/NP}\right)\ =\ \frac{1}{10}[10+2+4+6+8+9+10+11+11+12]\ =\ 8.3\ \ minutes$$

Figure 2.11.LPT/NP Schedule for Two Slow Printers (10,000 Lines for Job J)

All these calculations are summarized in the table below:

Table 2.3.Performance Measures(10,000 Lines for Job J)

Configuration	*Schedule*	*Mean Throughput Rate*	*Mean Flow Time*
One Slow Printer	SPT/NP	1 job every 2.3 minutes	6.5 minutes
One Slow Printer	LPT/NP	1 job every 2.3 minutes	17.5 minutes
Two Slow Printers	SPT/NP	1 job every 1.6 minutes	4.7 minutes
Two Slow Printers	LPT/NP	1 job every 1.2 minutes	8.3 minutes
One Fast Printer	SPT/NP	1 job every 1.15 minutes	3.9 minutes
One Fast Printer	LPT/NP	1 job every 1.15 minutes	8.75 minutes

2.2.5 Summary What lessons have we learned?

- A single printer mean throughput rate is less insensitive to scheduling compared with a distributed two printer system
- In order to achieve greater mean throughput rate, either we speed up a single printer or we add printers; how many and where depend upon the workload
- Responsiveness or mean flow time is critically influenced for any of these examples by the workload and the configuration
- One large job can impact the performance of the two printer system much more adversely than the single high speed printer system

2.3 A More General Example

N jobs are present at some initial time, say zero, and are executed on P identical processors. The execution times for the jobs are denoted by $T_K, K=1,...,N$. The jobs are independent of one another: there is no precedence ordering among the jobs. How long does it take to execute all the jobs? One way to answer this is to calculate the total execution time, T_{finish} or T_F, required for each possible ordering of the jobs; since there are N jobs, there are $N!$ schedules, and we will find out for moderate values of N such as 20 to 30 that even trying out one schedule a second can take us centuries to investigate all possible schedules. Our approach here is to find upper and lower *bounds* on the total time required to execute all N jobs on P processors without investigating all possible scheduling rules.

2.3.1 One Processor Used First, suppose we had P processors, but only used *one* processor. This is the *worst* we might do: it gives an upper bound on T_F equal to the sum of the execution times for all the jobs.

$$T_F \leq \sum_{K=1}^{N} T_K$$

Furthermore, this gives us a *lower* bound on mean throughput rate:

$$\textit{mean throughput rate} \geq \frac{N}{T_F} = \frac{N}{\sum_{K=1}^{N} T_K} = \frac{1}{T_{average}} \quad T_{average} = \frac{\sum_{K=1}^{N} T_K}{N}$$

While this is an example of one type of *upper* bound on the total make span or an equivalent *lower* bound on mean throughput rate, it is *not* the best possible set of bounds. How can we do better? A digression is needed.

2.3.2 The Geometry of Static Scheduling The figure below shows an illustrative mode of operation for this system: the number of busy processors versus time is plotted. Initially all processors are busy, until one by one they become idle and all the work is completed at T_F.

Figure 2.12.Illustrative Operation: Number of Busy Servers vs Time

We denote by $S(t)$ the number of busy servers at time t. The area under the curve formed by $S(t)$ is simply the total amount of processor time all the jobs require:

$$\int_0^{T_F} S(t)dt = \sum_{K=1}^{N} T_K$$

In the figure above, we have denoted by $\tilde{T}$ the duration of time from the last instant when all the processors are busy until all processors are first idle. Our program is to relate the model ingredients, i.e., the area under the curve $S(t)$, the number of processors, the job processing times, and T_F, by bounding $\tilde{T}$.

2.3.3 A Lower Bound on Makespan What is the *shortest* total execution time? Since we have P processors, they could all start execution at the same time and finish at the same time:

$$\frac{\sum_{K=1}^{N} T_K}{P} \leq T_F$$

On the other hand, if one job requires more execution time than the average amount of processor time per job, then this one job will determine the shortest possible make span:

$$\max_K T_K = T_{\max} \leq T_F$$

We can combine all these bounds:

$$\max \left[T_{\max}, \frac{\sum_{K=1}^{N} T_K}{P} \right] \leq T_F$$

If no one job requires significantly more execution time than any other job, then

$$T_{\max} \ll \frac{\sum_{K=1}^{N} T_K}{P}$$

and hence all processors are simultaneously executing work.

On the other hand, if one job requires significantly more execution time than any other job, then

$$T_{\max} \simeq \sum_{K=1}^{N} T_K$$

and effectively only one processor is busy executing work.

2.3.4 An Upper Bound on Makespan The largest that $\tilde{T}$ might be is the largest time required to run any single job:

$$\tilde{T} <= \max_K T_K = T_{\max}$$

On the one hand, we can *lower* bound the total area under this curve by demanding that all but one of the processors finish all their work at the same time, and the final processor is busy executing one job for $\tilde{T}$:

$$P[T_F - \tilde{T}] + \tilde{T} \leq \sum_{K=1}^{N} T_K$$

If we rearrange this, we see that

$$T_F \leq \frac{\sum_{K=1}^{N} T_K}{P} + \frac{P-1}{P}\tilde{T} \leq \frac{\sum_{K=1}^{N} T_K}{P} + \frac{P-1}{P} T_{\max}$$

We can rewrite this as

$$T_F \leq \frac{\sum_{K=1}^{N} T_K}{P}\left[1 + \frac{(P-1)T_{\max}}{\sum_{K=1}^{N} P_K}\right]$$

The first term is simply the average time per processor to execute jobs. If no one job requires much more processing time than any other job, i.e., if

$$(P-1)T_{\max} \ll \sum_{K=1}^{N} T_K$$

then the total time to execute all the jobs is roughly the total execution time per processor:

$$T_F \simeq \frac{\sum_{K=1}^{N} T_K}{P}$$

If one job requires much more processing time than any other job, i.e., if

$$T_{\max} \simeq \sum_{K=1}^{N} T_K$$

then the total time to execute all the jobs is roughly equal to the single processor execution time:

$$T_F \simeq \sum_{K=1}^{N} T_K$$

On the other hand, if one job takes virtually all the time, then effectively only one processor can be used, so this should not be that surprising.

2.3.5 Speedup At any given instant of time there are $J_E(t)$ jobs *in execution* on P processors. Since job K requires T_K time units to be executed, the total execution time of jobs must equal the *integral* or *area* of $J_E(t)$ from the initial time $t=0$ to the end of execution of all jobs T_F:

$$\sum_{K=1}^{N} T_K = \int_0^{T_F} J_E(t)dt$$

On the one hand, the total time required to execute all jobs with one processor $P=1$ is simply the sum of all the job execution times:

$$T_F(P=1) = \sum_{K=1}^{N} T_K$$

On the other hand, the definition of speedup in going from $P=1$ to $P>1$ processors is simply

$$speedup = \frac{T_F(P=1)}{T_F(P>1)}$$

Combining all of the above, the speedup can be written as

$$speedup = \frac{1}{T_F}\int_0^{T_F} J_E(t)dt = E[J_E]$$

$$mean\ number\ of\ jobs\ in\ execution = E[J_E] = speedup$$

This result is fundamental: if there is *one* resource (such as a single processor), there is *no* opportunity for speedup. If there are *multiple* resources (one processor, one disk, one printer, one terminal), there can be as many opportunities for multiplying the mean throughput rate as there are resources.

2.4 Preemptive vs Nonpreemptive Scheduling

What about *preemptive* versus *nonpreemptive* scheduling? Here we see that we can in fact *achieve* the lower bound on make span. Two cases arise: either there is one job that takes longer than the average time per processor of all the jobs, i.e.,

$$T_{finish} = \max_K T_K = T_{\max}$$

or there is no job that takes longer than the average time per processor:

$$T_{finish} = \frac{\sum_{K=1}^{N} T_K}{P}$$

Combining all this, we see

$$T_{finish} = \max\left[\max_K T_K, \frac{\sum_{K=1}^{N} T_K}{P}\right]$$

The figure below shows a *nonpreemptive* schedule for three processors with a fixed workload to make this concrete.

Preemption will allow us to achieve the *smaller* of these two bounds. How can this be achieved? One way is to assign jobs to the first processor until T_{finish} is passed, and then assign the overlap plus other jobs to the next processor until T_{finish} is passed, and so on until we assign all the work.

EXERCISE: How in fact can we achieve this schedule? It appears that we run the end of the job before we run its beginning.

EXERCISE: How do we circumvent the problem that no job can execute in parallel with itself?

In fact this bound is achievable if there is no precedence ordering among the jobs, i.e., some jobs must be done earlier than others.

Figure 2.13A.Three Processor Nonpreemptive Schedule

The figure below illustrates a preemptive schedule, and shows that the make span for the preemptive schedule equals the average processor time.

Figure 2.13B.A Preemptive Schedule for Three Processors

2.5 Impact of Partial Ordering and Fluctuations

Suppose we have a list of N jobs, and for convenience we suppose $N=2P$. We break the list of jobs into two lists, $J_1,...,J_P, J'_1,...,J'_P$. The jobs are not independent: some jobs must be executed before other jobs. We denote by $J'_K<J_K, K=1,...,P$ the constraint that J'_K must be executed before J_K can begin execution. Each unprimed job requires one unit of memory to be executed; each primed job requires P units of memory to be executed. The execution time for the unprimed jobs are all identical and equal to one; the execution time for each of the primed jobs is identical and equal to ε which we will make very small compared to one:

$$T_K = 1 \gg T'_K = \varepsilon \quad K=1,...,M$$

We have a total of P units of memory and P processors. We wish to investigate two different schedules:

- Schedule one: $J'_1,...,J'_M, J_1,...,J_M$
- Schedule two: $J_1,...,J_M, J'_1,...,J'_M$

For the first schedule, in order to satisfy the partial order constraint, we execute the primed jobs in order, and then execute in parallel on the M processors all the unprimed jobs: The make span or total time to finish all the jobs is

$$T_{finish} = P\varepsilon + 1 \quad \textit{schedule one}$$

For the second schedule, in order to satisfy the partial order constraint, we execute a primed job and then an unprimed job in pairs until we execute all the jobs: The make span or total time to finish all the jobs is

Figure 2.14.Schedule One for Three Processors

Figure 2.15.Schedule Two for Three Processors

$$T_{finish} = P(1+\varepsilon) \quad schedule\ \ two$$

The ratio of these two can vary immensely. For example if ε=1 then

$$\frac{T_{finish,I}}{T_{finish,II}} = \frac{P(1+\varepsilon)}{1+P\varepsilon}\Big|_{\varepsilon=1} = \frac{2P}{P+1} \underset{P\to\infty}{\to} 2$$

and hence we can be off by no more than a factor of *two* with these two schedules. On the other hand, if $\varepsilon \ll 1$, then

$$\frac{T_{finish,I}}{T_{finish,II}}\Big|_{\varepsilon\ll 1} = \frac{P(1+\varepsilon)}{1+P\varepsilon} \underset{\varepsilon\to 0}{\to} P$$

and hence we can be off by a factor of P which could be *much* greater than just a factor of two!

2.6 Polynomial Evaluation

Job precedence constraints arise in evaluation of polynomials, which is frequently done in practice in signal processing. These give us additional concrete examples to build intuition concerning the performance of parallel processors. Our problem is to evaluate a polynomial Y:

$$Y = \sum_{K=0}^{N} A_K X^K$$

The inputs are the coefficients $A_K, K=0,...,N$ and the value of X, while the output is the scalar Y. Given

P processors, we want to understand what is the *minimum* time required to completely evaluate one such polynomial. We will do so in special cases, to build insight.

2.6.1 Addition We begin with the case where all the coefficients are arbitrary, but $X=1$. We must evaluate Y where

$$Y = \sum_{K=1}^{N} A_K$$

Each addition is assumed to take one time unit. If we had one processor, this would require $N-1$ additions, and hence $N-1$ time units. Given four processors, and N=15, the table below shows the operations of each step of the evaluation.

Table 2.4.Summation Evaluation with Four Processors

Time	*Processor 1*	*Processor 2*	*Processor 3*	Processor 4
1	$Z_1=A_0+A_1$	$Z_2=A_2+A_3$	$Z_3=A_4+A_5$	$Z_4=A_6+A_7$
2	$Z_5=A_8+A_9$	$Z_6=A_{10}+A_{11}$	$Z_7=A_{12}+A_{13}$	$Z_8=A_{14}+Z_1$
3	$Z_9=Z_2+Z_3$	$Z_{10}=Z_4+Z_5$	$Z_{11}=Z_6+Z_7$	IDLE
4	$Z_{12}=Z_8+Z_9$	$Z_{13}=Z_{10}+Z_{11}$	IDLE	IDLE
5	$Y=Z_{12}+Z_{13}$	IDLE	IDLE	IDLE

For four processors, the total time required to sum fifteen coefficients is now five, while for one processor the total time was fourteen. The speedup is the ratio of these two:

$$speedup = \frac{T_{finish}(P=1)}{T_{finish}(P>1)} = \frac{14}{5} = 2.8 \qquad N=15$$

The best possible speedup would be to keep all processors busy, and hence this would be a factor of four. Effectively, we have 4–2.8=1.2 idle processors.

What happens as $N \to \infty$? Now all four processors are continuously busy, and hence

$$speedup = \frac{T_{finish}(P=1)}{T_{finish}(P>1)} = P=4 \qquad N \to \infty$$

What happens as $P \to \infty$? At the first step, each processor will add one term to another term, reducing the total number of items by a factor of two. This can be repeated, until $\log_2(N)$ time steps elapse for summing all terms together.

Combining all these ideas, it can be shown that

$$T_{finish} \geq \left\lceil \frac{N}{P} \right\rceil - 1 + \left\lceil \log_2[min(P,N)] \right\rceil$$

where $\lceil X \rceil$ is the smallest integer greater than or equal to X.

EXERCISE: Show that for $P=4, N=15$ this lower bound on T_{finish} is achieved.

2.6.2 Powers Suppose that $A_N=1, A_K=0, K \neq N$, so we wish to evaluate $Y=X^N$. If one processor is available, and multiplication requires one unit of time, then to evaluate Y worst case would require $N-1$ time steps. However, we can do better! Suppose that $N=2^5$, i.e., N is a power of two. Instead of taking thirty one time steps to evaluate Y, consider the following procedure:

Table 2.5.Steps for Evaluating $Y=X^{32}$

Step	*Value*
1	$Z_1=X^2$
2	$Z_2=Z_1^2=X^4$
3	$Z_3=Z_2Z_2=X^8$
4	$Z_4=Z_3Z_3=X^{16}$
5	$Y=Z_4Z_4=X^{32}$

If $N \to \infty$, it can be shown in general that even if N is not a power of two a single processor can evaluate X^N in $\log_2(N)$ multiplications:

$$T_{finish} \approx \log_2(N) \quad N \to \infty, P=1$$

On the other hand, for $P \to \infty$, i.e., with an infinite number of processors, P processors can evaluate X^N in $\log_2(N)$ multiplications:

$$T_{finish} \approx \log_2(N) \quad P \to \infty$$

Hence, *one* processor can evaluate X^N as quickly as an *infinite* number of processors, and we can gain *nothing* by parallelism.

2.6.3 General Case In general, we wish to evaluate Y where

$$Y = \sum_{K=0}^{N} A_K X^K$$

For $P=1$, a single processor, one algorithm for evaluating Y is given by

$$Y = ((..((A_N X + A_{N-1})X + A_{N-2})X + A_{N-3})...)X + A_0$$

For $N=22$ this requires twenty five additions and multiplications for $P=1$.

On the other hand, for $P=2$, one algorithm for evaluating Y is given by

$$Y = \sum_{K=0}^{N/2} A_{2K} X^{2K} + X \sum_{K=0}^{N/2-1} A_{2K+1} X^{2K+1}$$

and hence we must evaluate two polynomials in X^2. For $N=22$ and $P=2$ this requires twenty four additions and multiplications.

For $P=3$, one algorithm for evaluating Y involves writing Y as the sum of three polynomials in X^3, and using all three processors.

Combining all these items, it can be shown that

$$\frac{2N}{P} \log_2(P) \leq T_{finish} \leq \frac{2N}{P} \log_2(P) + o(\log_2(P))$$

The final term, $o(log_2(P))$, is negligible as $P \to \infty$, in the sense that

$$\lim_{P \to \infty} \frac{o(log_2(P))}{P} = 0$$

Hence, the speedup, measured in the time required to evaluate this expression with P processors versus $P=1$ is roughly given by

$$speedup = \frac{T_{finish}(P=1)}{T_{finish}(P>1)} \approx \frac{P}{\log_2(P)} \quad P \to \infty$$

2.6.4 Summary We have shown that

- Evaluating roughly N binary operations with P processors can lead to a speedup approaching P
- Evaluating X^N with P processors leads to *no* speedup over a single processor
- Evaluating a polynomial in N terms with P processors leads to a speedup of $P/log_2(P)$ over a single processor, which is in between the other two cases.

The lesson here: the workload can significantly impact the actual benefit of using multiple processors.

2.6.5 Additional Reading

[1] I.Munro, M.Paterson, *Optimal Algorithms for Parallel Polynomial Evaluation,* Journal of Computer System Science, **7,** 189 (1973).

[2] A.H.Sameh, *Numerical Parallel Algorithms--A Survey*, in **High Speed Computer and Algorithm Organization,** D.J.Kuck, D.H.Lawrie, A.H.Sameh (editors), pp.207-228, Academic Press, NY, 1977.

2.7 Critical Path Scheduling

P parallel processors are available for executing jobs fed from a single queue. First, suppose there is no precedence ordering among jobs, i.e., no job need be done before any other job. One scheduling rule called *critical path* scheduling is to execute tasks according to their processing time, with longest tasks first. The intuitive notion is that the longest jobs are *critical* in determining how short the make span can be, and hence it is essential to be operating on the critical path schedule for shortest make span or highest mean throughput rate. A different way of thinking about this is to allow the number of processors P to become infinite: each job will be assigned to one processor, and the make span equals the time to do the longest job. With a finite number of processors the make span will be longer than with an infinite number of processors, and hence we have a *lower* bound on the total time to finish a workload.

If there is a precedence ordering of jobs, then we sort times required for each job *stream* longest to shortest, and schedule the first job as the job that is the start of the longest critical path or longest stream that must be executed, and having scheduled this first job we now repeat this exercise with one less job, until all jobs are scheduled.

EXERCISE: Construct a flowchart for critical path scheduling, and exhibit pseudo code for implementing the flow chart.

Again, a different way of thinking about this is to allow the number of processors P to become infinite: each job will be assigned to one processor, and the make span equals the time to do the longest path of jobs. Viewed in this way, we see that

$$\max\left[T_{critical\ path}, \frac{1}{P}\sum_{K=1}^{N} T_K\right] \le T_{finish}(N)$$

where $T_{critical\ path}$ is the make span for a critical path schedule.

$$T_{finish}(N) \le \frac{1}{P}\sum_{K=1}^{N} T_K \left[1 + \frac{(P-1)T_{critical\ path}}{\sum_{K=1}^{N} T_K}\right]$$

If the make span for a critical path schedule is much less than the average time each processor is busy, then

$$T_{finish}(N) \approx \frac{1}{P}\sum_{K=1}^{N} T_K$$

and hence the make span is reduced by P or the mean throughput rate increases by P.

On the other hand, if the make span for a critical path schedule is much more than the average time each processor is busy, then

$$T_{finish}(N) \approx T_{critical\ path}$$

and effectively there is no gain with more than one processor.

In a different vein, it can be shown that

$$\frac{T_{finish,CRITICAL\ PATH}}{\min\limits_{SCHEDULE} T_{finish,SCHEDULE}} \le \frac{4}{3} - \frac{1}{3P}$$

In other words, critical path scheduling is no worse than any other single processor (P=1) scheduling rule in minimizing T_{finish}, is 1/6 longer than the *best* (in terms of minimizing T_{finish}) two processor scheduling rule, and is 1/3 longer than the best infinite processor scheduling rule!

2.8 P Identical Parallel Processors

In practice, the execution times of the jobs are not often known with any precision. This occurs for a variety of reasons: the data to be manipulated varies from job to job, the text program executes different branches depending upon the data, and so forth. Here we attempt to capture this phenomenon by fixing the mean or average execution time of a job, and modeling the fluctuations of job execution times as non negative random variables drawn from the same distribution.

A more precise statement of the problem is as follows: N jobs must be executed by P servers or processors. The jobs are all present at an initial time, say zero. The time to execute job K=1,...,N is denoted by T_K. We will assume that the sequence of job execution times are non negative random variables, with the same marginal distribution:

$$PROB[T_K \leq X] = G(X) \quad K=1,2,...,N$$

The mean time to execute a job is denoted by $E[T]$ where

$$E[T] = \int_0^\infty X dG(X) = \int_0^\infty [1 - G(X)]dX < \infty$$

The time to completely execute all N jobs is denoted by C_N.

The mean throughput rate of completing jobs is given by the ratio of the total number of jobs divided by the mean time required to complete all the jobs:

$$mean\ \ throughput\ \ rate = \frac{N}{E[C_N]}$$

The shortest that C_N could be is

$$C_N \geq \frac{1}{S} \sum_{K=1}^{N} T_K$$

The longest that C_N could be is

$$C_N \leq max[T_{\max}, \frac{1}{S} \sum_{K=1}^{N} T_K]$$

where

$$T_{\max} = \max_{K=1,...,N} T_K$$

The mean throughput rate is the ratio of the total number of jobs divided by the mean time to complete these jobs:

$$mean\ \ throughput\ \ rate \equiv \lambda_N = \frac{N}{E[C_N]}$$

Our goal is to allow the number of jobs to become larger and larger, $N \to \infty$, such that the mean throughput rate stabilizes at an average or limiting value. In fact, this limit is

$$\lim_{N \to \infty} \lambda_N = \frac{P}{E[T]}$$

In words, each job requires a mean execution time of $E(T)$, and we will realize a *speedup* of P because all the processors will be busy. As a bonus, the upper and lower bounds on mean throughput rate will also approach this limit under these conditions. This shows that the *details* of the workload may not matter nearly so much as might be expected.

2.8.1 Analysis For any real number, say Y, we will use the following trick:

$$T_{\max} \leq Y + \sum_{K=1}^{N} U_{-1}[T_K - Y]$$

where $U_{-1}(X) = 0\ X<0, = 1\ X>0$ is a so called generalized unit step function. This allows us to write

$$T_{\max} \le Y + N \int_{Y}^{\infty} [1 - G(X)]dX$$

We denote by ω the small value of X such that $1 - G(X)=0$:

$$\omega = \inf_{X} \{X \mid 1 - G(X) = 0\}$$

Two cases arise:

2.8.2 ω finite For this case, $\omega<\infty$, we see that

$$T_{\max} \le \omega \quad \rightarrow \quad E[T_{\max}] \le \omega$$

and hence

$$\frac{P}{E[T]} \frac{1}{1 + \frac{(P-1)\omega}{NE(T_{\max})}} \le \lambda_N \le \frac{P}{E[T]}$$

As the number of jobs becomes larger and larger, $N \rightarrow \infty$, the mean throughput rate as well as upper and lower bounds approach the desired result:

$$\lim_{N \rightarrow \infty} \lambda_N = \frac{P}{E[T]}$$

2.8.3 ω infinite For this case, we find it useful to define a related quantity μ_N:

$$\mu_N = \inf_{X>0} \left\{ X \mid 1 - G(X) \le \frac{1}{N} \right\}$$

Because $\omega=\infty$, it is clear that $\lim_{N \rightarrow \infty} \mu_N = \infty$.

On the other hand, we see that

$$E[T_{\max}] \le \mu_N + N \int_{\mu_N}^{\infty} [1 - G(X)]dX$$

Because

$$\lim_{N \rightarrow \infty} \int_{\mu_N}^{\infty} [1 - G(X)]dX = 0$$

we see that

$$N \int_{\mu_N}^{\infty} [1 - G(X)]dX = o(N)$$

This implies that

$$1 - F(\mu_N) \le \frac{1}{N} \quad \rightarrow \quad 1 - G(X) = o\left[\frac{1}{X}\right]$$

This in turn implies

$$o\left[\frac{1}{\mu_N}\right] \le \frac{1}{N}$$

and in turn that

$$\mu_N = o(N)$$

Finally, we see that

$$E[T_{\max}] = o(N)$$

and this allows us to show that

$$\lim_{N\to\infty} \lambda_N = \frac{P}{E[T]}$$

as in the previous case.

2.8.4 An Example Suppose we examine a particular $G(X)$:

$$G(X) = 1 - Qexp[-QX/E[T]] \quad 0<Q\leq 1$$

This has a mean at $E[T]$ and a great deal of fluctuation about the mean, with the fluctuations increasing as $Q\to 0$.

The trick here is to fix $1 - G(\mu_N)$ at $1/N$:

$$1 - G(\mu_N) = Qexp[-Q\mu_N/E[T]] = \frac{1}{N}$$

This in turn fixes μ_n:

$$\mu_N = \frac{E[T]\ ln(NQ)}{Q}$$

Now if we substitute into the earlier expressions, we see

$$E[T_{\max}] \leq E[T] \left[\frac{ln(NQ)}{Q} + \frac{1}{Q}\right]$$

Finally, the mean throughput rate is upper and lower bounded by

$$\frac{P}{E[T]} \frac{1}{1 + \frac{(P-1)[1+ln(NQ)]}{NQ}} \leq \lambda_N \leq \frac{P}{E[T]}$$

2.8.5 Bounds The longest possible completion time C_N occurs when we execute all jobs but one, and the remaining job requires the largest amount of processing time of all jobs:

$$C_N \leq \frac{1}{P}\sum_{K=1}^{N-1} T_K + (T_N = T_{\max}) \equiv C_{\max}$$

The shortest possible completion time C_N occurs when either all jobs finish execution at the same instant of time on all P processors, or all but one job are executed on $P-1$ processors and the remaining job executes on one processor and has the largest processing time:

$$C_N \geq \max[T_{\max}, \frac{1}{P}\sum_{K=1}^{N} T_K] = C_{\min}$$

The ratio of the longest to the shortest completion times is upper bounded by

$$\frac{C_{\max}}{C_{\min}} \leq 2 - \frac{1}{P}$$

In other words, as we go from P=1 to P=2 processors, the greatest *relative* change in the mean completion time for *any* scheduling policy is $2 - 1/2 = 1.5$, while going from P=2 to P=3 processors gives a maximum gain due to scheduling of $2 - 1/3 = 5/3$.

2.9 Single Processor Deadline Scheduling

In deadline scheduling, different classes of tasks have different urgencies; we denote by W_K the allowable queueing time window that we can tolerate for job $K=1,...,N$. At the arrival epoch of a job, say job K, that arrives at time A_K we look up its window in a table, add the window to its arrival time, and call the result the *deadline* or priority or urgency number for that task:

$$deadline\ for\ job\ K \equiv D_K = A_K + W_K \quad K=1,2,...,N$$

That job is inserted in a queue in priority order or deadline order, most urgent jobs first, with jobs executed in deadline order. Note that windows need not be positive! We compare performance via queueing time statistics and via a different pair of quantities. If we have two classes of jobs, each with its own window, and we fix the *difference* between the two windows but allow the *smaller* (or larger) window to become infinite in size, then we approach *static* priority scheduling. Very roughly speaking, static priority scheduling allows control over first moments of queueing times of tasks, while deadline and other dynamic scheduling policies allow control of asymptotic distributions of queueing and waiting times. The measures of performance of interest are

- *lateness*-- the lateness of job K is defined as its completion time minus its deadline;

  $$L_K = C_K - D_K \quad K=1,2,...$$

  Negative lateness implies the job queueing time was less than its deadline, while positive lateness means the job queueing time exceeded its deadline.

- *tardiness*-- the tardiness of job K is zero if the job is finished by its deadline, and equals the lateness otherwise; put differently, the tardiness is the maximum of zero and the lateness

  $$T_K = max[0,L_K] \quad K=1,2,...$$

What are desirable properties of deadline scheduling? One is that deadline scheduling minimizes the *maximum* lateness or *maximum* tardiness over *any* scheduling policy. This suggests that deadline scheduling will be useful in time critical jobs, which is perhaps not so surprising! How do we show this property? Suppose we had a schedule that violated deadline scheduling ordering yet had a smaller maximum lateness than deadline scheduling. This means that there is at least one time interval where one job, say job J, is waiting and another job, say job I, is executing, even though the deadline for job J is less than the deadline for job I. We denote the non deadline schedule by S' while the deadline schedule is denoted by S. Since the maximum lateness is smaller using S' we see

$$max[L(S')] \leq max[max[L(S)],L(S')]$$

Since the completion time of job I under schedule S' is given by

$$C_I(S') = max[C_J(S),C_I(S)]$$

we see that

$$max[L(S')] \leq max[max[L(S),C_J(S)-D_I,L_I(S)]$$

Finally, since the deadline for I is greater than that for J, we see

$$max[L(S')] \leq max[max(L(S)),L_J(S),L_I(S)] = max[L(S)]$$

This is precisely what we wanted to show.

Several consequences follow immediately:

- Even knowing the arrival pattern in advance cannot help minimize the maximum lateness better than deadline scheduling
- If the windows are equal to the service times for the respective jobs, then deadline scheduling minimizes the maximum waiting time
- If all the windows are equal to one another, then deadline scheduling is equivalent to service in order of arrival

Suppose we have two types of jobs, A and B, with processing times and windows as shown:

Table 2.6.Job Statistics Summary

Job Type	*Service*	*Window*
A	T_A=3 seconds	W_A=5 seconds

B T_B=2 seconds W_B=3 seconds

The figure below shows an arrival pattern consisting of one type A arrival at time t=0 followed by a type B arrival at time t=1 and t=4, with the resulting work pattern for deadline scheduling, and static priority scheduling (A higher priority than B, B higher priority than A).

Figure 2.16.Illustrative Comparison of Deadline vs Static Priority Scheduling

Note that *neither* static priority schedule can meet *all* deadlines, while the deadline schedule can. Hence the need for *dynamic* rather than *static* priority scheduling!

Suppose we had N distinct types of jobs, with each job type having its own service time T_K with associated window W_K and maximum storage for job K of B_K bytes, where K=1,...,N. A natural measure of utilization is given by U where U is defined to be the storage multiplied by the service time, and divided by the window, summed over all job types:

$$U = \sum_{K=1}^{N} \frac{B_K T_K}{W_K}$$

If $U<1$ then we want to show that all deadlines can be met. Let's examine the worst case, where the maximum number of requests are always present and the service time is the largest possible. This implies that server is busy handling type K requests at a rate of B_K/W_K. Since the requests are absorbing the maximum possible processing times, the utilization of the server will equal U; but for the entire system to keep up with its work, we must demand $U<1$.

Example: Suppose we had a set of sources (memory boards, disk controller boards, terminal controller boards, processor boards, and so on) that each demand access to a shared bus. If each source can only have one request outstanding at one time, and we choose the window equal to the service time for each type of bus access request, and all the service demands will be met if and only if

$$\sum_{K=1}^{N} \frac{1}{T_K} \leq 1$$

2.10 Pipelining

Suppose that each processor is dedicated to executing a given function. One special case of this is to construct a *pipeline* of processors, where the input of one processor is the output of another processor, and so forth:

Figure 2.17.Three Stage Pipeline

Each job consists of a series of steps, that must be done in exactly this order.

2.10.1 An Example Six jobs each require the following processing at each of three steps:

Table 2.7.Execution Times for Six Jobs

Job	*Step 1*	*Step 2*	*Step 3*
1	3	2	4
2	2	4	1
3	1	5	2
4	3	2	2
5	1	1	1
6	2	1	4

The schedule for this set of jobs is (1,2,3,4,5,6), i.e., execute the jobs in the order of job number. A schedule is shown in the figure below:

2.10.2 A More Sophisticated Example Suppose N jobs must be executed on a $P=N$ stage pipeline. The execution times of job step I for job $K=1,...,N$ is denoted by T_{IK}. Suppose that the execution times are given by

$$T_{IK} = \varepsilon, \quad I \neq K \qquad T_{IK}=1, \quad I=K$$

Two different schedules are of interest:

- Schedule one: $(1,2,...,N)$
- Schedule two: $(N,N-1,...,1)$

For schedule one, the figure below shows an illustrative plot of the activity of each processor: The total time required to execute all the work for P processors is

Figure 2.18.Three Stage Pipeline Schedule For Six Jobs

Figure 2.19.Schedule One Processor Activity(P=3,N=3)

$$T_{finish,I} = P + (P-1)\varepsilon$$

For schedule two, the figure below shows an illustrative plot of processor activity:

Figure 2.20.Schedule Two Processor Activity(P=3,N=3)

The total time required to execute all the work for P processors is

$$T_{finish,II} = 1 + (P+1)\varepsilon$$

There are two cases of interest: $\varepsilon=1$ so all jobs take the same amount of time, and hence

$$T_{finish,I} = T_{finish,II} = P$$

and $\varepsilon \ll 1$, so that only stage K of job J_K requires one time unit of processing, and hence

$$T_{finish,I} \simeq P \qquad T_{finish,II} \simeq 1 \qquad \varepsilon \to 0$$

If there is radical imbalance, the time required to execute all the work can differ by the number of

processors!

2.11 Bounding The Make Span for Pipelines of Single Processors

Our goal in this section is to find upper and lower bounds on the make span for a pipeline of single processors, much as we did earlier for parallel processors.

Suppose a list or schedule of N jobs is to be followed, with the jobs ordered *(1,2,...,N).* P processors are available. Each job consists of $S=P$ steps. Each processor is dedicated to executing one and only one step. The time required to execute step $I=1,...,S$ of job $K=1,...,N$ is denoted by T_{IK}.

2.11.1 Upper Bound on Make Span One immediate upper bound on the total time required to execute all jobs is to simply execute the jobs one at a time:

$$T_{finish} \leq \sum_{K=1}^{N} \sum_{I=1}^{S} T_{IK}$$

EXERCISE: Can you find a *sharper* upper bound that is less than or equal to this upper bound and includes the same model parameters?

2.11.2 Lower Bound on Make Span In order to lower bound the make span, the total time required to execute all jobs, we realize that there are three possible contributions to the make span:

- The time required to execute job one from step one through step $J-1$: $\sum_{S=1}^{J-1} T_{1S}$
- The time required to execute every job at step J: $\sum_{K=1}^{N} T_{KJ}$
- The time required to execute job N from step $J+1$ through step S: $\sum_{I=J+1}^{S} T_{NI}$

Combining all these, we see that for one particular step in the pipeline, J,

$$\sum_{I=1}^{J-1} T_{1I} + \sum_{K=1}^{N} T_{KJ} + \sum_{I=J+1}^{S} T_{NI} \leq T_{finish}(J)$$

To get the best possible lower bound, we should look for the largest this set of lower bounds could be, for all steps:

$$\max_{1 \leq J \leq S} \left[\sum_{I=1}^{J-1} T_{1I} + \sum_{K=1}^{N} T_{KJ} + \sum_{I=J+1}^{S} T_{NI} \right] \leq T_{finish}$$

This development can be made more formal as follows. Suppose that C_{KJ} denotes the completion time of job step $J=1,...,S$ for job $K=1,...,N$. For convenience, we assume there is a fictitious initial stage, labeled zero, with $C_{K0}=0$ for all K. The completion times of step J of job one obey the following recursion:

$$C_{J1} = C_{J-1,1} + T_{J1} \quad J=1,...,S$$

The completion times of step J of job K obey the following recursions:

$$C_{KJ} = \max[C_{K,J-1}, C_{K-1,J}] + T_{KJ} \quad J=1,...,S$$

If we use the inequalities $X<max[X,Y]$ and $Y<max[X,Y]$ then we see

$$C_{KJ} \geq C_{K-1,J} + T_{KJ}$$

The completion time of step J of job N is similarly given by

$$C_{NJ} \geq C_{N,J-1} + T_{NJ}$$

Combining all of these bounds, we obtain the result sketched earlier.

2.12 Johnson's Rule

Now we turn to a two stage pipeline.

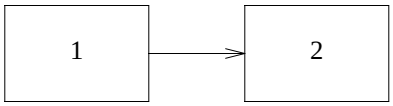


Figure 2.21.Two Stage Pipeline

N jobs have associated processing times $T_{1,K}$ at stage one and $T_{2,K}$ at stage two, $K=1,...,N$. Our goal is to minimize the total system busy time to completely execute all this work.

As an example, suppose we have five jobs (A, B, C, D, E) that must each be executed in two stages, with the execution times for each job summarized as follows:

Table 2.8.Job Execution Times

Job	*Stage 1*	*Stage 2*
A	6	3
B	0	2
C	3	4
D	8	6
E	2	1

The jobs are executed according to an alphabetical priority ordering. The total time to execute all the jobs according to this schedule, the make span, is 22.

Can you find a priority list schedule that achieves this?

The principal result here, due to Johnson, is

- Execute tasks on the second stage in the same order as on the first stage
- J_K is executed before J_I if

$$\min[T_{K,1},T_{I,2}] \le \min[T_{K,2},T_{I,1}]$$

An algorithm for finding a scheduling that does this is

- Find the minimum $T_{I,K}$ where $I=1,2$ and $K=1,...,N$
- If I=1 put this at the head of the schedule; if I=2 put it at the end of the schedule
- Delete this task from the schedule and repeat the above procedure until no tasks remain

Roughly speaking, we want to put tasks requiring big processing times at the start of the schedule and small processing time tasks at the end of the schedule. Now, how bad can we do? It can be shown that

$$\frac{T_{finish,SCHEDULE}}{\min\limits_{SCHEDULE} T_{finish,SCHEDULE}} \le P$$

or in other words, we could be off by the total number of processors in the pipeline!

2.13 S Stage Pipeline

Earlier, when we were analyzing parallel processor groups, we remarked that the workload is not precisely known in practice. We modeled that by allowing the mean execution time for each job to be fixed, and the fluctuations about the mean were modeled by random variables. Here we carry out exactly the same exercise.

2.13.1 Problem Statement N jobs must be executed by a system of P processors. Each job consists of S steps that must be done one after another; each step consists of execution of a given amount of text code operating upon a different amount of data, depending on the nature of the job. The system consists of a pipeline of stages: stage K consists of P_K processors fed from a single queue. Each processor can

execute only one job at a time. The execution time for step K is denoted by $T_K, K=1,...,S$.

The completion time for all N jobs is denoted by C_N. The mean throughput rate is the number of jobs divided by the mean completion time:

$$\lambda_N = \frac{N}{E[C_N]}$$

Our goal is to show that the mean throughput rate approaches a limit as the number of jobs becomes larger and larger:

$$\lim_{N\to\infty} \lambda_N = \min_K \left[\frac{P_K}{E(T_K)} \right]$$

In words, the stage with the lowest maximum mean throughput rate, the so called *bottleneck* stage, will determine the *system* total maximum mean throughput rate. Furthermore, the upper and lower bounds on mean throughput rate approach this value. Hence, the details of the workload do not matter as much as might be expected.

2.13.2 Upper Bound on Mean Throughput Rate One upper bound on mean throughput rate is given by the number of jobs in the system:

$$\lambda_N = \frac{N}{\sum_{K=1}^{N} E[T_K]}$$

A second upper bound on mean throughput rate is given by all the processors at a given stage being completely busy:

$$\lambda_N = \min_{K=1,...,S} \frac{P_K}{E[T_K]}$$

Combining all this, we see

$$\lambda_N \le \min \left[\min_{K=1,...,S} \frac{P_K}{E[T_K]}, \frac{N}{\sum_{K=1}^{N} E[T_K]} \right]$$

If the processing time at each stage has *no* fluctuations about the mean processing time, i.e., the processing times are deterministic, then we claim that this upper bound is *achievable.* Finally, as $N\to\infty$ we obtain the desired result.

2.13.3 Lower Bound on Mean Throughput Rate The longest completion time, and hence the lowest mean throughput rate, is given by executing only one job at a time:

$$E[C_N] \le N \sum_{K=1}^{S} \frac{E[T_K]}{P_K} + o(N)$$

If each service time is assumed to be an independent random variable, with

$$PROB[T_K \le X] = 1 - \alpha \exp\left[-\alpha X / E[T_K]\right] \quad K=1,...,S; X>0$$

so that the mean service time at stage K is fixed at $E(T_K), K=1,...,S$ but $\alpha\to 0$ results in greater and greater fluctuations about the mean.

In order to obtain our desired result, we must show that for any positive number, say $\varepsilon>0$, that

$$\lim_{\alpha\to 0} E[C_N] \ge N \sum_{K=1}^{S} \frac{E[T_K]}{P_K} - \varepsilon$$

To see this, we need introduce additional notation and machinery.

Let $X_1,...,X_N$ be nonnegative independent random variables with distribution functions given by $G_1(),..,G_N()$ respectively. We define $X_{\max}$ as the maximum of these random variables:

$$X_{\max} = \max[X_1,...,X_N]$$

Let E_K be the event that $X_K > Y$. The probability that the maximum $X_{\max}$ exceeds Y is given by

$$PROB[X_{\max} > Y] = PROB[\bigcup_{J=1}^{N} E_J]$$

This can be upper and lower bounded as follows:

$$PROB[\bigcup_{J=1}^{N} E_J] \leq \sum_{J=1}^{N} PROB[E_J]$$

$$PROB[\bigcup_{J=1}^{N} E_J] \geq \sum_{J=1}^{N} PROB[E_J] - \sum_{J<K} PROB[E_J \bigcap E_K]$$

If we substitute in, we see

$$\sum_{J=1}^{N} [1 - G_J(Y)] - \sum_{J<K} [1 - G_J(Y)][1 - G_K(Y)] \leq PROB[X_{\max} > Y]$$

$$PROB[X_{\max} > Y] \leq \sum_{J=1}^{N} [1 - G_J(Y)]$$

An alternate way of computing this event is to define Y_K as the total time that stage $K=1,..,S$ is not empty, and hence

$$C_N \geq \max[Y_1,...,Y_S]$$

On the other hand, each Y_K can be lower bounded by the total service time at stage K over the total number of processors at stage K

$$Y_K \geq \tilde{Y}_K \equiv \frac{total\ \ stage\ \ K\ \ service\ \ time}{P_K}$$

Now we realize that

$$PROB[\tilde{Y}_K \leq X] = G_K(X) = \sum_{J=0}^{N} \begin{bmatrix} N \\ J \end{bmatrix} \alpha^J (1-\alpha)^{N-J} \left[1 - \sum_{K=0}^{J-1} \tilde{E}_K \right]$$

$$\tilde{E}_K \equiv \frac{X^K}{K!} exp[-X] \qquad X = P_K \alpha X / E[T_K]$$

Hence, we see that

$$1 - G_K(X) = \sum_{J=1}^{N} \begin{bmatrix} N \\ J \end{bmatrix} \alpha^J (1-\alpha)^{N-J} \sum_{K=0}^{J-1} \tilde{E}_K$$

The mean of $\tilde{Y}_K$ is given by

$$E[\tilde{Y}_K] = \frac{NE[T_K]}{P_K}$$

while

$$1 - G_K(X) \leq constant(K)\ [1 - (1-\alpha)^N]$$

We see that

$$\lim_{\alpha \to 0} \sup_{X>0} [1 - G_K(X)] = 0$$

so that

$$E[C_N] \geq E[max(\tilde{Y}_1, \ldots, \tilde{Y}_S)]$$

$$\geq N \sum_{K=1}^{S} \frac{E(T_K)}{P_K} - \sum_{J<K} \int_0^\infty [1 - G_J(X)][1 - G_K(X)]dX$$

However, the last term can be made as small as possible by choosing α as close to zero as needed.

Finally, as we allow $N \to \infty$, we obtain the desired result for the mean throughput rate.

2.14 General Problem Classification

N jobs, labeled $J_K, K=1,...,N$ are to be executed. Each job can be processed on at most one processor at a time. This means that if a job can be processed on more than one processor at a time, we will break it up into one or more distinct jobs, each of which can be processed on only one processor at a time. Furthermore, each processor can execute at most one job at any instant of time.

2.14.1 Job Data Each job has its *release* time or *arrival* time, the earliest time it can begin execution, denoted by $A_K, K=1,...,N$. Each job has its *deadline* denoted by $D_K, K=1,...,N$ which is the time by which it should *ideally* be completed.

Each job requires a number of steps, with $S_K, K=1,...,N$ denoting the number of steps for $J_K, K=1,..,N$.

A *precedence relation* denoted by < between jobs may exist. We denote by $J_I < J_K$ the requirement that J_I is to complete before J_K can start. One way to show all of these is via a table, showing for each job step all the job steps that must be completed before that job step can be executed. A second way to show all of these is via a block diagram or *graph:* each job is a node in a graph, with arrows into each node emanating from job step nodes that must be completed *prior* to that job step. This is a *directed* graph: the arrows have direction. This graph has no *cycles:* there is no chain of arrows that completes a closed chain or cycle. This is an *acyclic* graph.

Example: Consider the following set of six jobs with a given precedence relationship:

Table 2.9.Six Job Precedence Relationships

This Job	*Must Be Preceded By*	*This Job*
2		1
3		1
4		3
5		2,6
7		4,5

The precedence relationships are summarized in the directed acyclic graph shown in Figure 2.22.

Figure 2.22.Precedence Constraints for Six Jobs

Each job can have its own *weight* to reflect the relative importance of a job: $W_K, K=1,...,N$. A nondecreasing real valued *cost* function, $F_K(t)$, measuring the cost incurred if J_K is completed at time t.

2.14.2 Resource Configuration There are P processors, with each processor being capable of executing a job step at a *different* rate. Each step of each job will require a given amount of processing time. Let T_{IKM} denote the amount of processing time required at step $I=1,...,S_K$ for job $K=1,...,N$ on processor $M=1,...,P$. If all processors are identical, we will *ignore* or suppress the subscript M due to different types of processors. If $T_{IKM}=\infty$ then we assume this step of this job will *never* by convention be executed on that processor (because it will take forever).

Each job will require one processor, which may be thought of as an *active* resource, and zero or more *passive* resources, such as memory or storage, an operating system file, and so forth. R_{IKL} denotes the amount of passive resources $L=0,...,\tilde{R}$ (possibly zero) required by job $J_K, K=1,...,N$ at step $I=1,...,S_K$. We assume that the total amount of resource type L available will not be exceeded by any one step of any one job.

2.14.3 Scheduling Policy The scheduling policy determines which job is executed at any given instant of time with a given set of resources. Competition for resources occurs *implicitly* via scheduling of resources, and *explicitly* via cooperation and precedence ordering between jobs.

We assume that if work is ready to be executed, that it will immediately be assigned to an idle processor (this rules out a variety of pathological situations from this point on).

Schedules can be *nonpreemptive* where once a job begins execution it runs to completion, or *preemptive* where once a job begins execution it can be interrupted by more urgent jobs. Preemptive scheduling can involve *resuming* execution at the point of interruption (and hence we call these schedules *preemptive resume)* or *repeating* execution anew (and hence we call these schedules *preemptive repeat)* The total amount of resource L available, $\tilde{R}_L, L=0,...,M$, cannot be exceeded at any instant of time by any allowable scheduling rule.

2.14.4 Performance Measures We will focus on a variety of performance measures for jobs.

Here are some examples of job oriented performance measures:

- The *completion time* for job J_K, denoted C_K
- The *lateness* for job J_K, denoted $L_K \equiv C_K - D_K$
- The *tardiness* for job J_K, denoted $T_K \equiv max[0, L_K]$

Here are some examples of system oriented performance measures:

- The total time required to execute all the jobs, T_{finish}, assuming all the ready times are *zero,* i.e., the jobs are all present at the same initial time. This is called the *make span* because it is the span of time it takes to make or execute all jobs
- The mean throughput rate, which is the total number of jobs executed divided by the total time interval

$$mean\ \ throughput\ \ rate = \frac{N}{T_{finish}}$$

- The fraction of time each processor is busy *at all* or its *utilization*
- The fraction of time two different processors are *simultaneously* busy

EXERCISE: Can you think of any more?

2.14.5 Parallelism and Concurrency Two jobs are said to be executing *concurrently* at a given time t if both started execution before time t and *neither* has completed execution. An example might be two programs that have been submitted by different people to be link edited and compiled on the same processor and are waiting for the compilation to be completed: the processor is *concurrently* executing each job, but at any given instant of time *one* job is using the single processor. This is called *logical*

concurrency.

Two jobs are said to be executing in *parallel* at a given time *t* if both are actively being executed or moving to completion at time *t.* An example would be a computer system with a single processor and a single secondary storage unit: two units can be executing in parallel, one using the processor and one using the secondary storage unit, at the same instant of time, providing the operating system supports this mode of operation. This is called *physical* concurrency.

2.14.6 Additional Reading

[1] R.L.Graham, E.L.Lawler, J.K.Lenstra, A.H.G.Rinnooy Kan, *Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey,* Annals of Discrete Mathematics, **5,** 287-326(1979).

[2] E.G.Coffman, Jr. (editor), **Computer and Job Shop Scheduling Theory,** Wiley, NY, 1976.

[3] M.S.Bakshi, S.R.Arora, *The Sequencing Problem,* Management Science, **16,** B247-263 (1969).

2.15 A Packet Switching System

A computer communication system receives packets from any of L lines and transmits packets over the same L lines. Two types of packets can be received: data packets and control packets. Control packets are required to set up a communication session between a transmitter and receiver pair, to acknowledge proper receipt of W data packets, and to conclude a communication session between a transmitter and receiver pair. Data packets are made up of pieces of a message stream between a transmitter and receiver. Each transmitter and receiver session demands one logical or passive resource, a *virtual circuit* that will be set up over a single *physical circuit* to allow time division multiplexing or sharing of the physical circuit among multiple pairs of transmitters and receivers. Two types of processors are available called slow and fast: the slow processor requires less buffering for each data packet than the fast processor, and is less expensive. For simplicity, the ready time of each control and data packet is assumed to be zero, i.e., all packets are available at time zero. Control packets have a deadline or urgency of ten milliseconds, while data packets have a deadline or urgency of one hundred milliseconds. A nonpreemptive schedule is used: once execution is begun, a packet is processed to completion. The table below summarizes the information needed to specify the performance of this system:

Table 2.10.Job Steps

Step	*Job*	*Type*	*Can Be Preempted By*
1	Control	Startup	Step 2
2	Control	Takedown	No Job
3	Data	Transmit	Step 1,2

The resources required are summarized below:

Table 2.11.Job Step Resource Requirements

Job	*Slow Processor*		*Fast Processor*		*Passive Resources*	
Step	*Time*	*Buffer*	*Time*	*Buffer*	*VC*	*Packet ID*
1	5 msec	32 Bytes	2 msec	64 Bytes	1	1
2	7 msec	32 Bytes	4 msec	64 Bytes	1	1
3	10 msec	512 Bytes	5 msec	1024 Bytes	1	1

The following information is available concerning the performance *goals* of the system:

Table 2.12.Packet Switching System Performance Goals

Step	*Criterion*	*Normal Business Hour*	*Peak Business Hour*
1	Window	50 msec	200 msec
2	Window	25 msec	100 msec
3	Window	100 msec	500 msec
1	Weight	10	100
2	Weight	20	500
3	Weight	1	10
1	Cost	1	5
2	Cost	2	10
3	Cost	2	8

Finally, the precedence ordering for jobs is as follows

Table 2.13.Job Step Precedence Ordering

This Step	*Must Be Preceded By*	*This Step*
2		1,3
3		1

EXERCISE: Compare the performance during a normal and peak business hour of a static priority schedule with priority ordering (2,1,3) with a deadline priority schedule.

2.15.1 Additional Reading

[1] R.W.Conway, W.L.Maxwell, L.W.Miller, **Theory of Scheduling,** Chapters 1-7, Addison Wesley, Reading, Mass., 1967.

[2] E.G.Coffman (editor); J.L.Bruno, E.G.Coffman, Jr., R.L.Graham, W.H.Kohler, R.Sethi, K.Steiglitz, J.D.Ullman (coauthors), **Computer and Job-Shop Scheduling Theory,** Wiley, New York, 1976.

[3] R.L.Graham, *Combinatorial Scheduling Theory,* pp.183-211, in **Mathematics Today: Twelve Informal Essays,** edited by L.A.Steen, Springer-Verlag, NY, 1978.

[4] M.R.Garey, R.L.Graham, *Bounds for Multiprocessor Scheduling with Resource Constraints,* SIAM J.Computing, **4,** 187-200 (1975).

[5] M.R.Garey, R.L.Graham, D.S.Johnson, *Performance Guarantees for Scheduling Algorithms,* Operations Research, **26,** 3-21 (1978).

[6] D.J.Kuck, *A Survey of Parallel Machine Organization and Programming,* Computing Surveys, **9** (1), 29-59 (1977).

[7] J.T.Schwartz, *Ultracomputers,* ACM Transactions on Programming Languages and Systems, **2** (4), 484-521 (1980).

[8] G.R.Andrews, F.B.Schneider, *Concepts and Notations for Concurrent Programming,* Computing Surveys, **15** (1), 3-43 (1983).

Problems

1) Two N tuples denoted by $\underline{X}=(X_1,...,X_N)$ and $\underline{Y}=(Y_1,...,Y_N)$ are inputs to a computer system. The output of the system is the scalar inner product, denoted $<\underline{X},\underline{Y}>$, of the two inputs:

$$<\underline{X},\underline{Y}> = \sum_{K=1}^{N} X_K Y_K$$

P identical processors are used to evaluate the inner product expression. Each processor is capable of executing one addition or one multiplication in one second.

A. For $P=1$ and $N=16$ find a schedule that minimizes the total time, T_{finish}, required to evaluate one inner product $<\underline{X},\underline{Y}>$.

B. For $P=4$ processors and $N=16$ find a schedule that minimizes T_{finish} to evaluate one inner product $<\underline{X},\underline{Y}>$. Compute the speedup in going from one to four processors.

C. Repeat all the above for $N=17$

D. If the number of inputs is a power of two, i.e., $N=2^J, J=1,2,..$ and the number of processors is a power of two, i.e., $P=2^K, K=1,2,..$, show that the total time T_{finish} required to evaluate an inner product need not exceed

$$\left\lceil \frac{2N}{P} \right\rceil - 1 + \left\lceil \log_2(P) \right\rceil$$

where

$$\lceil X \rceil = \textit{smallest integer greater than or equal to } X$$

2) A computer system takes as input a sixteen tuple $(A_1,..,A_{16})$ and generates a scalar output X:

$$X = A_1(A_2 + A_3(A_4 + A_5(A_6 +.. + A_{13}(A_{14} + A_{15}A_{16}))))))$$

A. One processor is used to evaluate X. The binary operations of operations of addition and multiplication each take one unit of time to perform. How long does it take to evaluate X?

B. With two processors, show that it is possible to evaluate X in ten units of time.

3) Nine jobs are present at time zero. Each job requires one processor for the amount of time shown below:

Table 2.14.Job Execution Time

Job	*Time*	*Job*	*Time*	*Job*	*Time*
1	3	4	2	7	4
2	2	5	4	8	4
3	2	6	4	9	9

Each job requires one processor for execution; no job can execute in parallel with itself. Nonpreemptive scheduling is used: once a job begins execution, it runs to completion. These jobs are not independent, but have a precedence ordering:

Table 2.15.Job Precedence

This Job	*Must Be Preceeded By*	*This Job*
9		1
5,6,7,8		4

A. Construct a schedule for three processors. What is the minimum time required to complete all nine jobs?

B. Repeat part (A) for two processors

C. Repeat part (A) for four processors

D. Repeat part (A) with three processors but all the execution times reduced by one

E. Repeat part (A) with three processors but the precedence constraint is now weakened as shown in the table below:

Table 2.16.Job Precedence Summary

This Job	*Must Be Preceded By*	*This Job*
9		1
7,8		4

4) You have just been put in charge of an assembly line for bicycle manufacturing. The first thing you learn is that assembling a bicycle is broken up into a number of specific smaller jobs:

- **FP--** Frame preparation, including installation of the front fork and fenders
- **FW--** Mounting and front wheel alignment
- **BW--** Mounting and back wheel alignment
- **DE--** Attaching the derailleur to the frame
- **GC--** Attaching the gear cluster
- **CW--** Attaching the chain wheel to the crank
- **CR--** Attaching the crank and chain wheel to the frame
- **RP--** Mounting right pedal and toe clip
- **LP--** Mounting left pedal and toe clip
- **FA--** Final attachments (including mounting and adjustment of handlebars, seat, brakes)

Each step takes a person a given number of minutes, summarized in the table below:

Table 2.17.Job Step Summary

Job	FP	FW	BW	DE	GC	CW	CR	RP	LP	FA
Time(Minutes)	7	7	7	2	3	2	2	8	8	18

Certain jobs must be done before others: try mounting the front fork to the bicycle frame if the brake cables are already attached! The table below summarizes which jobs must precede which others during assembly:

Table 2.18.Job Precedence Summary

This Job	*Must Be Preceded By*	*These Jobs*
FA		FP,FW,BW,GC,DE
BW		GC,DE,FP
FW		FP
GC,CW		DE
LP,RP		CR,CW,GC
CR		CW

Because of space and equipment constraints, the twenty assemblers are paired into ten teams of two people each. The goal is to have each team assemble fifteen bicycles in one eight hour shift. The factory quota is one hundred fifty bicycles assembled in one eight hour shift. For a team of two assemblers, the standard priority schedule has been FP, DE, CW, CR, GC, FW, BW, LP, FA, RP. This means that each assembler scans this list of work, highest priority on down, until a job is found that can be done while meeting the precedence constraints. The assembler will work on this job until it is finished, with no interruptions.

A. Plot the activity of each assembler on a team versus time for each bicycle. Will the factory meet its quota?

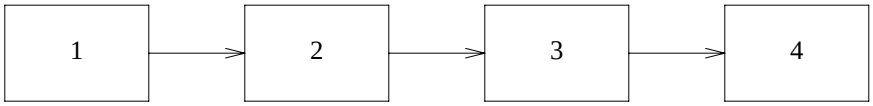
B. You rent all electric power tools for all assemblers. This reduces the time to do each job by one minute. For the standard schedule plot the activity of each assembler versus time for one bicycle. Will the factory meet its quota?

C. You return the rented tools, and hire a third assembler for each team. For the standard schedule, plot the activity of each assembler versus time for one bicycle. Will the factory meet its quota?

D. For two assemblers per team, using a critical path schedule, plot the activity of each assembler versus time for one bicycle. Will the factory meet its quota?

E. **BONUS:** Repeat B),C) using a critical path schedule. Will the factory meet its quota?

5) Four processors are arranged in a pipeline to execute a stream of jobs.



Four Stage Processor Pipeline

Six jobs are executed on this system, and require the following processing at each step:

Table 2.19.Processing Times per Step

Job	*Step 1*	*Step 2*	*Step 3*	*Step 4*
1	4	4	5	4
2	2	5	8	2
3	3	6	7	4
4	1	7	5	3
5	4	4	5	3
6	2	5	5	1

A. Construct a schedule for executing the six jobs on this processor pipeline.

B. Compute the mean flow time for the above schedule:

$$E(F) = \frac{1}{N}\sum_{K=1}^{N} F_K \quad N=6$$

C. Let $J(t)$ denote the number of jobs in the system, either waiting to be executed or in execution, at time t. Calculate the mean number of jobs in the system, $E(N)$:

$$E(N) = \frac{1}{T_F}\int_0^{T_F} J(t)dt$$

D. Show that

$$E(N) = \frac{N}{T_F}\, E(F)$$

6) During passage through a computer system, a job is processed by two different processors in sequence. We denote by the ordered pair (T_1,T_2) the processing time required at the first stage T_1 and at the second stage T_2 by each job. Assume that six (6) jobs arrive simultaneously for service. The processing times for this workload are given by (4,2), (2,3), (3,1), (3,1), (6,4), (2,4). Construct a schedule which processes the six jobs in the minimum time.

7) A system consists of P identical processors, each of which can execute *IPS* maximum assembly language instructions per second. This system must execute N identical tasks, all present at an initial time say zero, and each of which comprises N_{inst} assembly language instructions. The time to execute one task on one processor is $T_{P=1}$ and is given by

$$N_{inst} = T_{P=1}\, IPS \quad \rightarrow \quad T_{P=1} = \frac{N_{inst}}{IPS}$$

With P processors, the time required to execute a single task is T_P. The *speedup* is defined as the ratio of the single to multiple processor (single task) execution times:

$$speedup = \frac{T_{P=1}}{T_P}$$

A. If $\pi(K),K=1,...,P$ denotes the fraction of time K processors are simultaneously active executing work, show that

$$speedup = mean\ number\ of\ busy\ processors = \sum_{K=1}^{P} \pi(K)K$$

B. Let $N_{inst}(K)$ denote the number of instructions executed by K simultaneously active processors. Show that

$$N_{inst}(K) = \pi(K)T_P\ [K\ IPS] \quad K=1,2,...,P$$

C. With P processors, F(K), K=1,...,P, denotes the fraction of assembly language instructions that are executed concurrently or in parallel on K processors. Show that

$$F(K) = \frac{N_{inst}(K)}{\sum_{J=1}^{P} N_{inst}(J)} \quad K=1,2,..,P$$

D. Show that

$$speedup = \frac{1}{\sum_{K=1}^{P} \frac{F(K)}{K}}$$

E. We wish to evaluate repetitively sums consisting of fifteen terms:

$$Y = A_1 + \cdots + A_{15}$$

for different choices of $A_K, K=1,...,15$. We have four processors, (P=4); each processor can evaluate one partial sum at a time. The total sum can be evaluated in five steps as follows:

Table 2.20.Four Processor Evaluation of Fifteen Term Summations

Step	Processor One	Processor Two	Processor Three	Processor Four
1	$B_1=A_1+A_2$	$B_2=A_3+A_4$	$B_3=A_5+A_6$	$B_4=A_7+A_8$
2	$B_5=A_9+A_{10}$	$B_6=A_{11}+A_{12}$	$B_7=A_{13}+A_{14}$	$B_8=A_{15}+B_1$
3	$C_1=B_2+B_3$	$C_2=B_4+B_5$	$C_3=B_6+B_7$	IDLE
4	$D_1=B_8+C_1$	$D_2=C_2+C_3$	IDLE	IDLE
5	$Y=D_1+D_2$	IDLE	IDLE	IDLE

The intermediate scratch values are denoted by $B_1,...,B_8,C_1,\ldots,C_3$ and D_1,D_2 in the steps above. Find $\pi(K),F(K);K=1,2,3,4$. Compute the speedup factor directly from the table above, and from $\pi(K)$ and $F(K)$ directly.

F. For F(K)=1/P, show that

$$speedup = \frac{P}{\sum_{K=1}^{P}\frac{1}{K}} \approx \frac{P}{ln(P+1)+\gamma} \quad as \quad P\to\infty$$

$$where \ \gamma = Euler's \ constant = 0.5772156649$$

G. Verify that the algorithm for evaluating fifteen term summations with four processors obeys the following formula:

$$maximum \ speedup = \frac{P}{\frac{P}{N-1}\left\lceil\frac{N}{P}\right\rceil - \frac{P}{N-1} + \frac{P\log_2[min(N,P)]}{N-1}}$$

$$\lceil x \rceil = smallest \ integer \ greater \ than \ or \ equal \ to \ x$$

It can be shown* that if we wish to evaluate N degree polynomials on P processors,

$$Y = \sum_{K=0}^{N} A_K X^K$$

$$maximum \ speedup = \frac{P}{\log_2 P}\ \frac{1}{1+\frac{P\ F(P)}{2N\log_2 P}} \qquad \lim_{P\to\infty}\frac{F(P)}{\log_2 P}\to 0$$

8) Messages are processed by a transmitter and then a receiver. The order of processing messages at the transmitter and receiver is identical. The processing time for message K by the transmitter is denoted by T_K, and by the receiver is denoted by R_K. There are a total of N messages to be transmitted at time zero. We define $T_{N+1}=0$, $R_0=0$ for simplicity. The receiver can buffer at most two messages at any one time; once the receiver has two messages, the transmitter stops processing messages until the receiver only has one message.

---

* I.Munro, M.Paterson, *Optimal Algorithms for Parallel Polynomial Evaluation,* Journal of Computer System Science, **7,** 189 (1973).

A. What are the precedence relations for processing N messages?

B. Show that the total time or make span to process all N messages is given by

$$make\ span = T_F = \sum_{K=0}^{N} \max \left[T_{K+1}, R_K \right]$$

C. Show that the total time or make span to process all N messages can be written as

$$make\ span = T_{finish} = \sum_{K=1}^{N} (T_K + R_K) - \sum_{K=1}^{N-1} \min \left[T_{K+1}, R_K \right]$$

D. The mean throughput rate is defined as

$$mean\ throughput\ rate = \lim_{N \to \infty} \frac{N}{T_{finish}}$$

Show that

$$mean\ throughput\ rate = \frac{1}{E[\max(T,R)]}$$

$$E[max(T,R)] \equiv \lim_{N \to \infty} \frac{1}{N} \sum_{K=1}^{N} max(T_{K+1}, R_K)$$

where $E[max(T,R)]$ is the average of the maximum of the transmitter and receiver time per message. For $T_K = T = constant, R_K = R = constant$, explicitly evaluate this expression. For $R = 1$ and $T = 1, 0.5, 0.2$ what is the mean throughput rate?

E. Show that

$$T_{finish} \geq \max \left[R_N + \sum_{K=1}^{N} T_K, T_1 + \sum_{K=1}^{N} R_K \right]$$

F. What changes if the receiver can buffer an infinite number of messages?

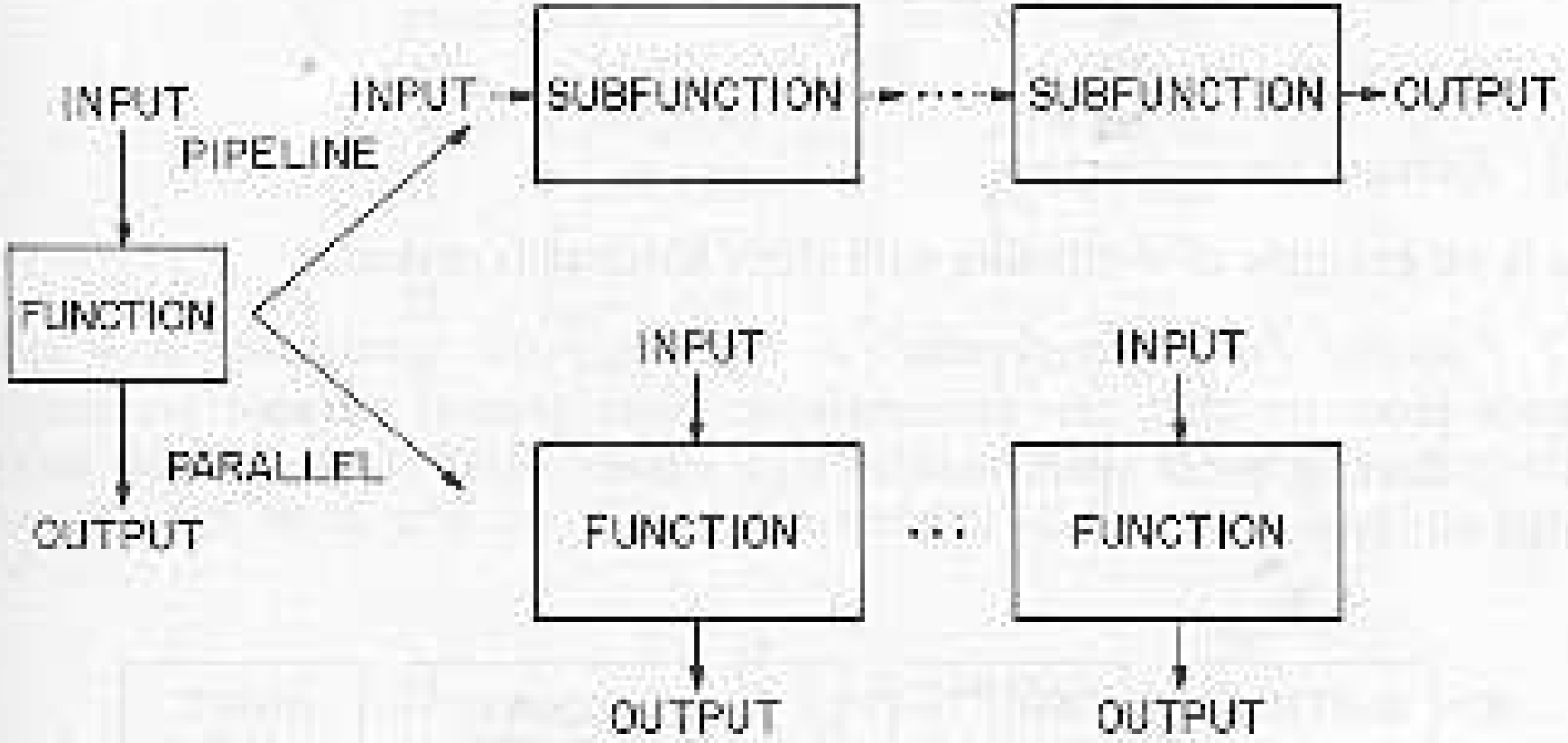


Figure 2.1.Pipeline and Parallel Configurations

Figure 2.2.Arithmetic Logic Unit Block Diagram

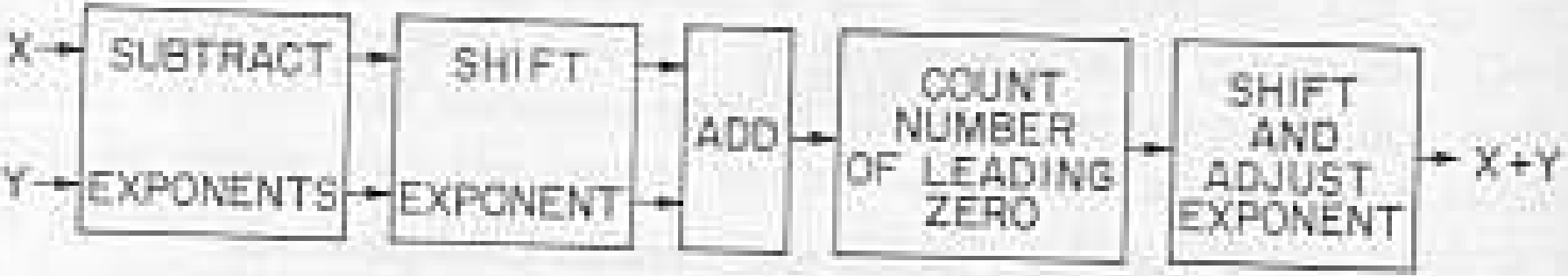


Figure 2.3.Floating Point Accelerator Block Diagram

Figure 2.4.SPT/NP Single Slow Printer Schedule

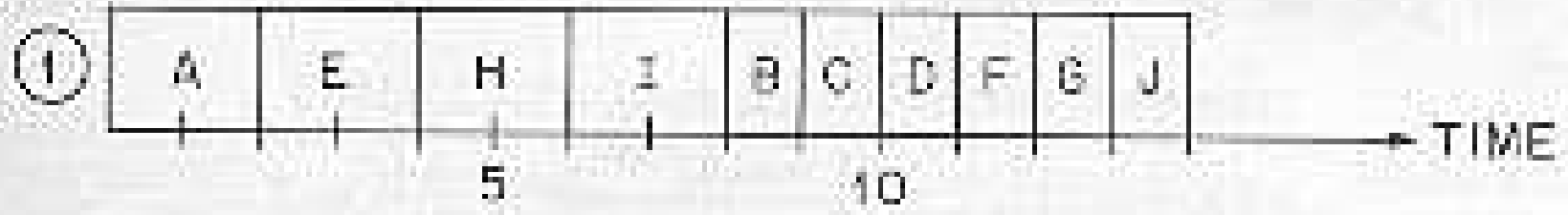


Figure 2.5.1.PT/NP Single Slow Printer Schedule

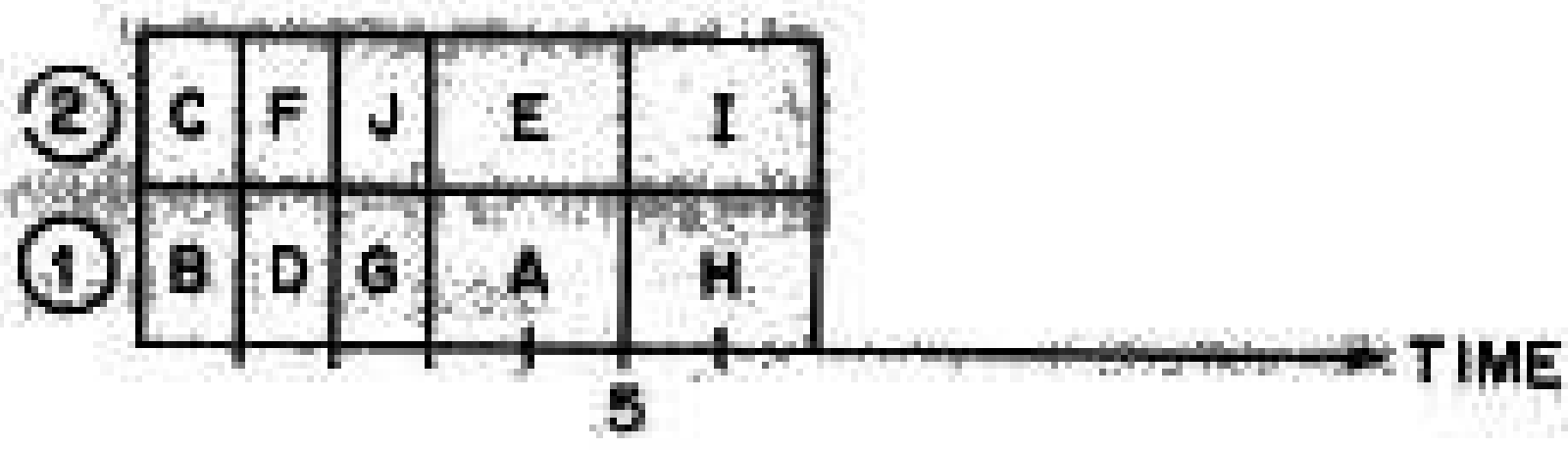


Figure 2.6.SPT/NP Two Slow Printer Schedule

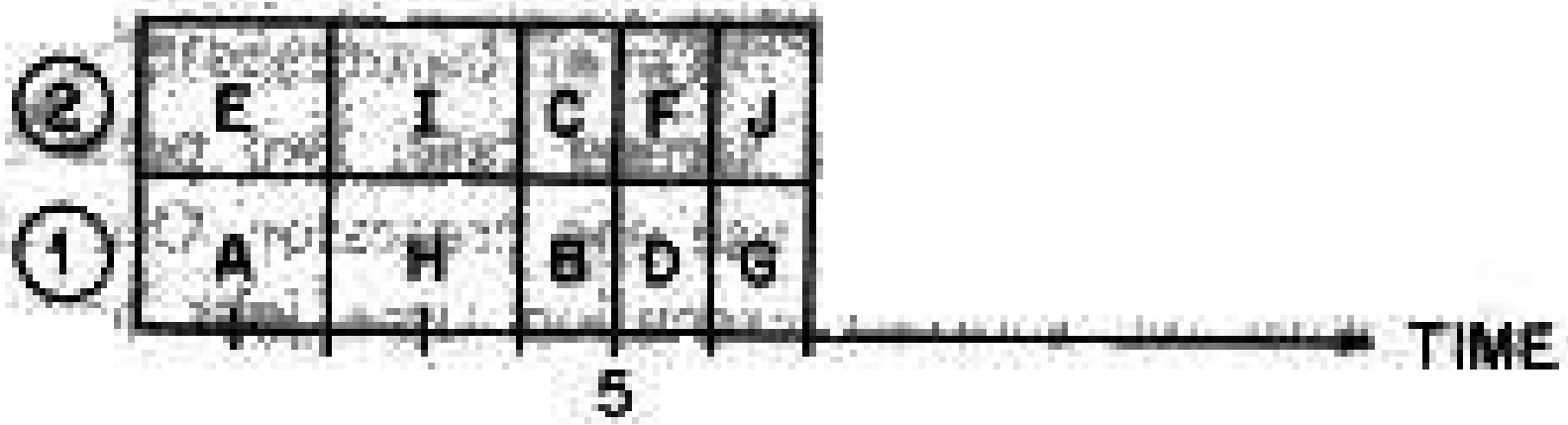


Figure 2.7.LPT/NP Two Slow Printer Schedule

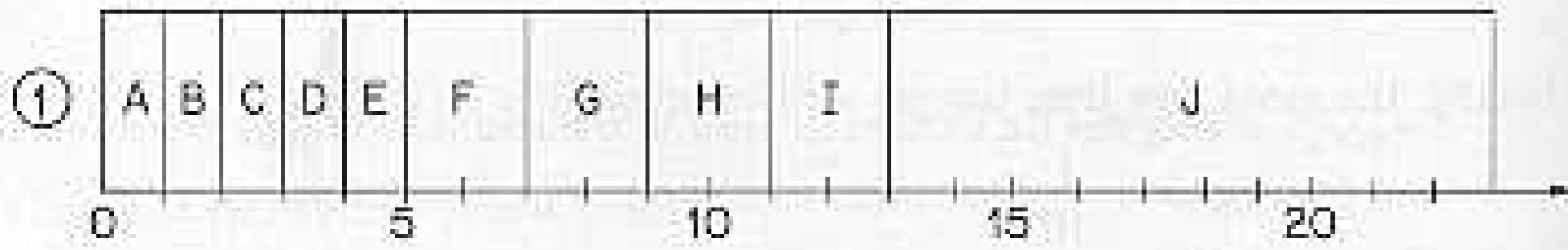


Figure 2.8.SPT/NP Schedule for One Slow Printer (10,000 Lines for Job J)

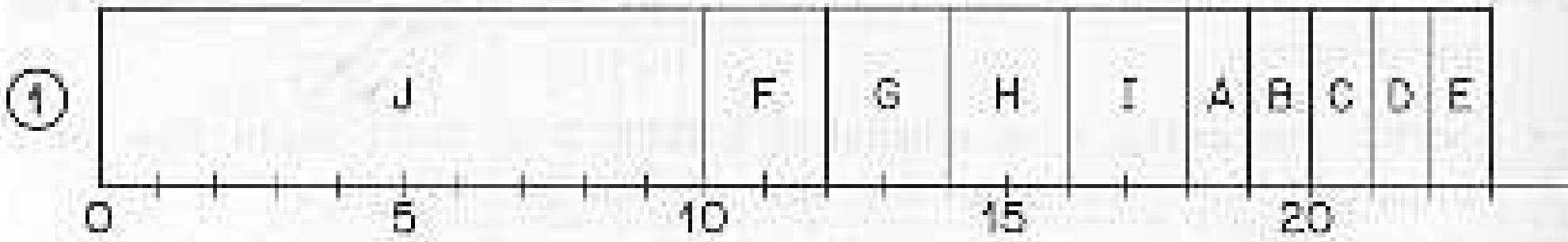


Figure 2.9.LPT/NP Schedule for One Slow Printer (10,000 Lines for Job J)

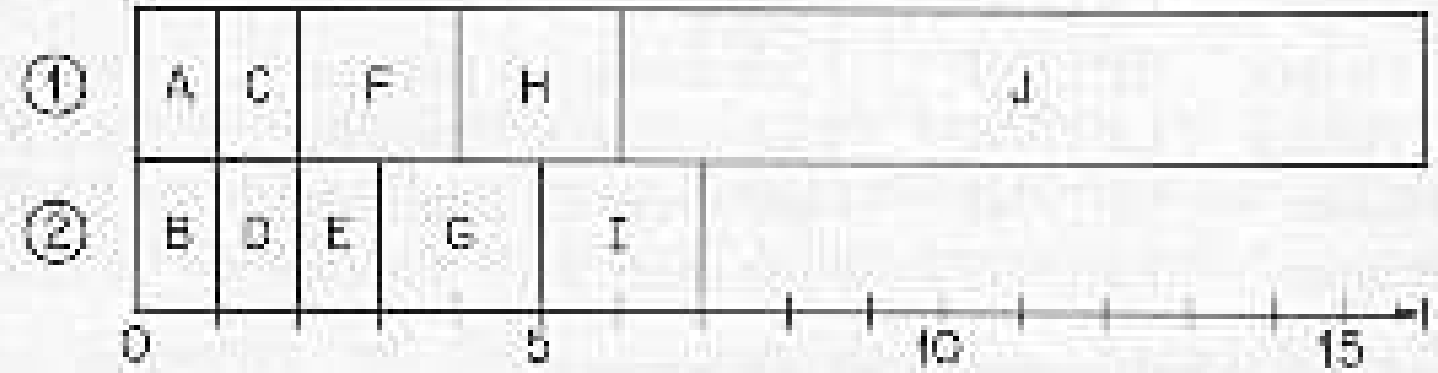


Figure 2.10.SPT/NP Schedule for Two Slow Printers (10,000 Lines for Job J)

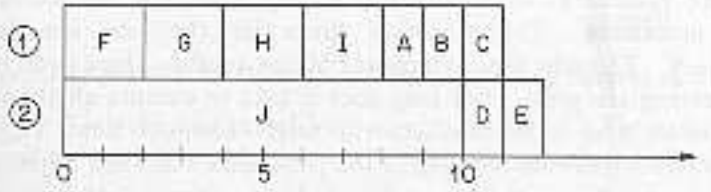


Figure 2.11.1.PT/NP Schedule for Two Slow Printers (10,000 Lines for Job J)

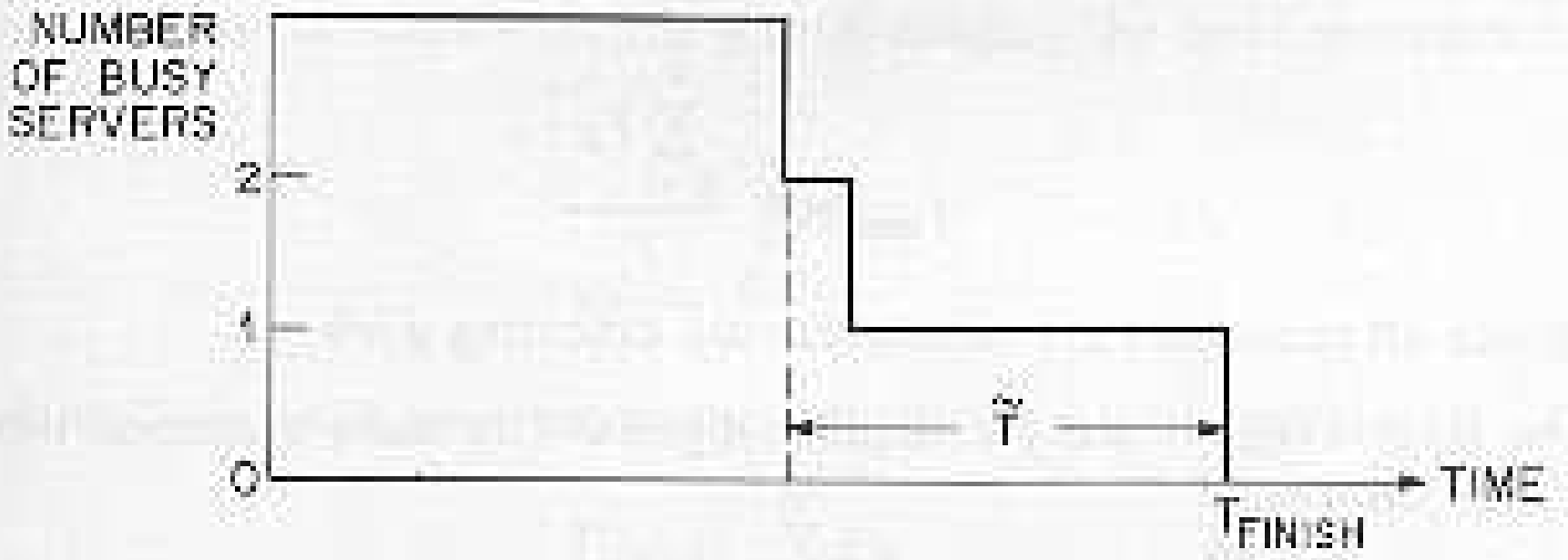


Figure 2.12.Illustrative Operation: Number of Busy Servers vs Time

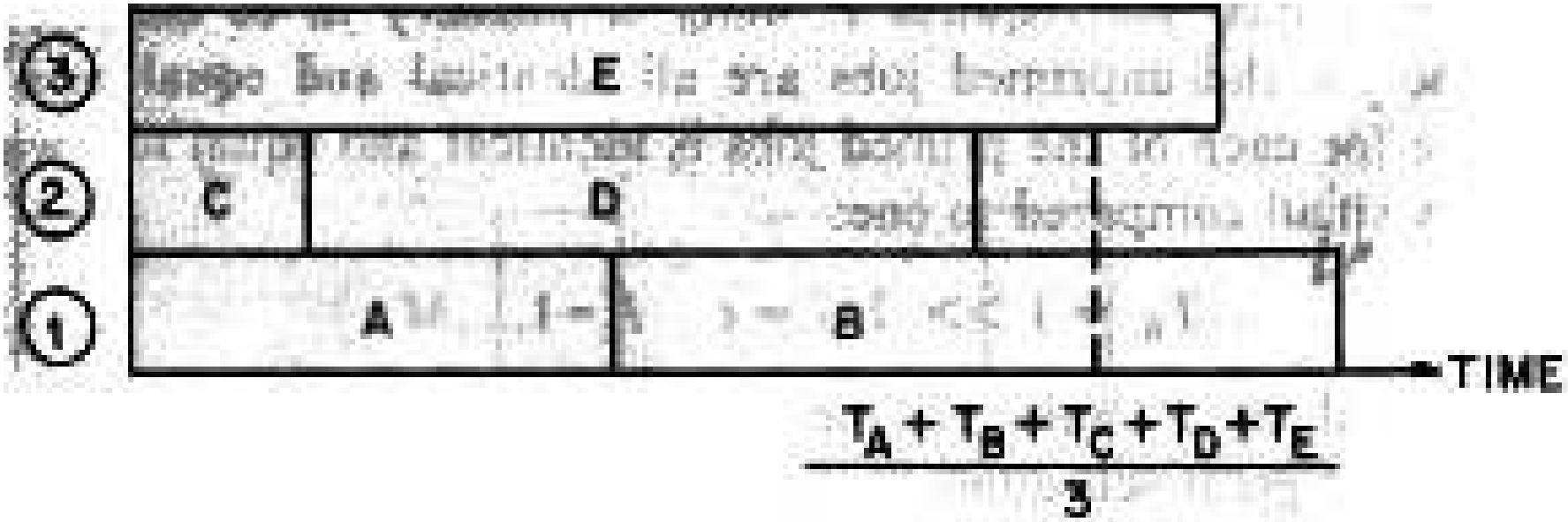


Figure 2.13A.Three Processor Nonpreemptive Schedule

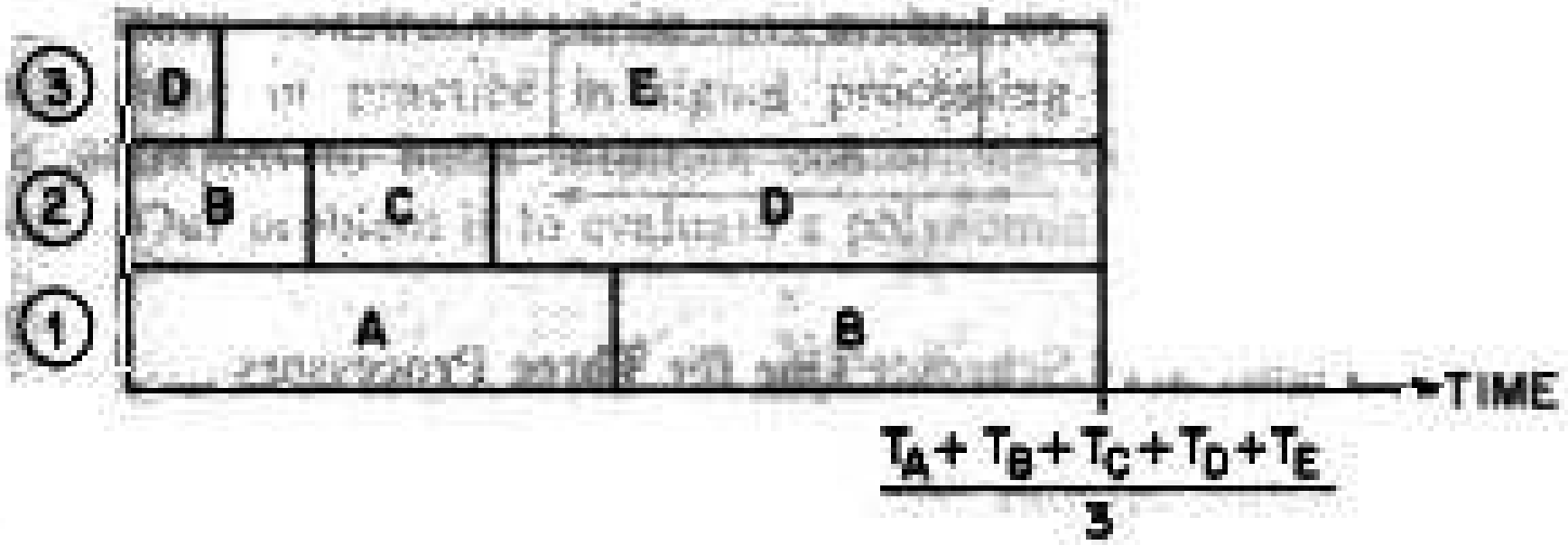


Figure 2.13B.A Preemptive Schedule for Three Processors

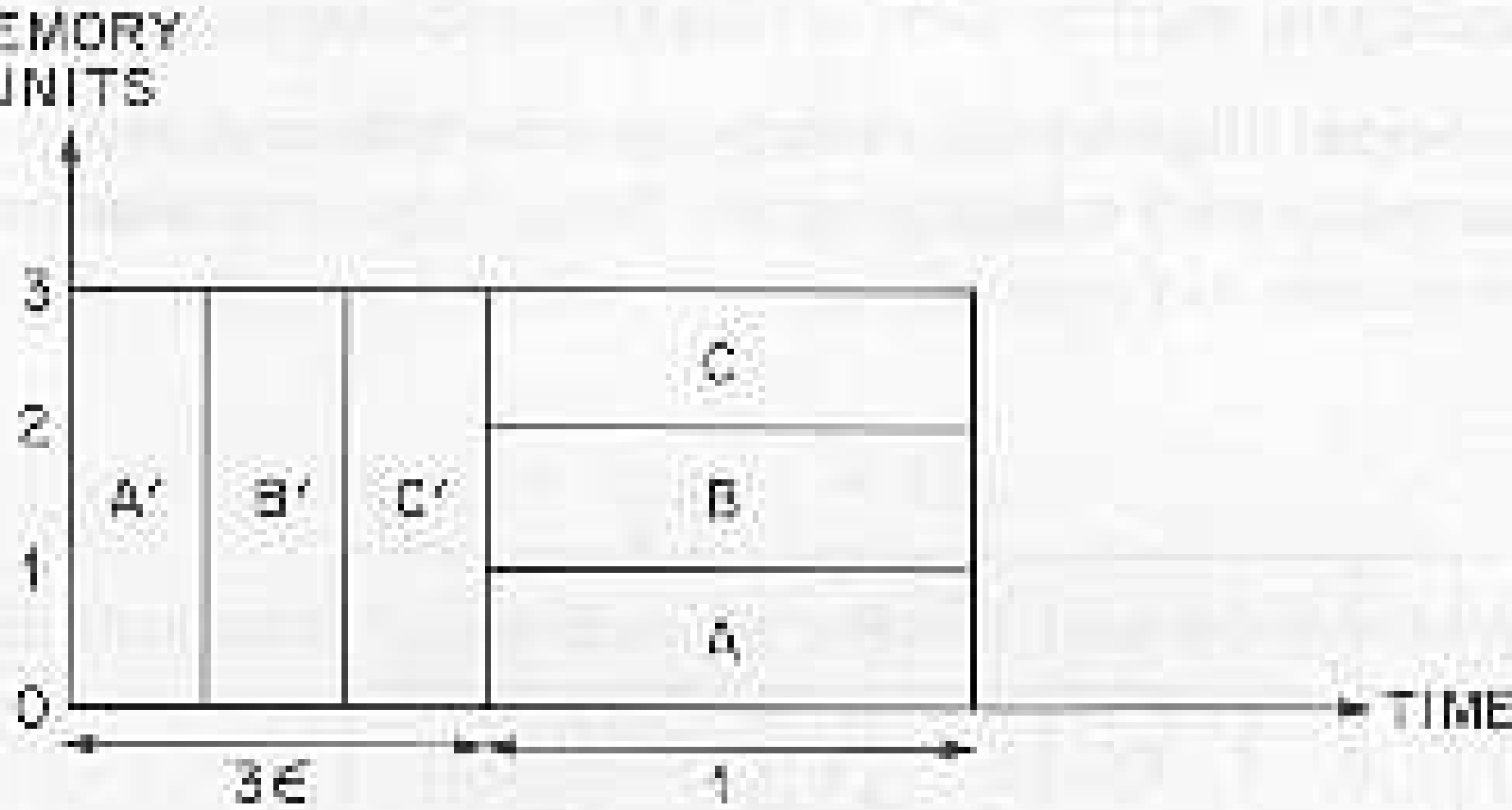


Figure 2.14.Schedule One for Three Processors

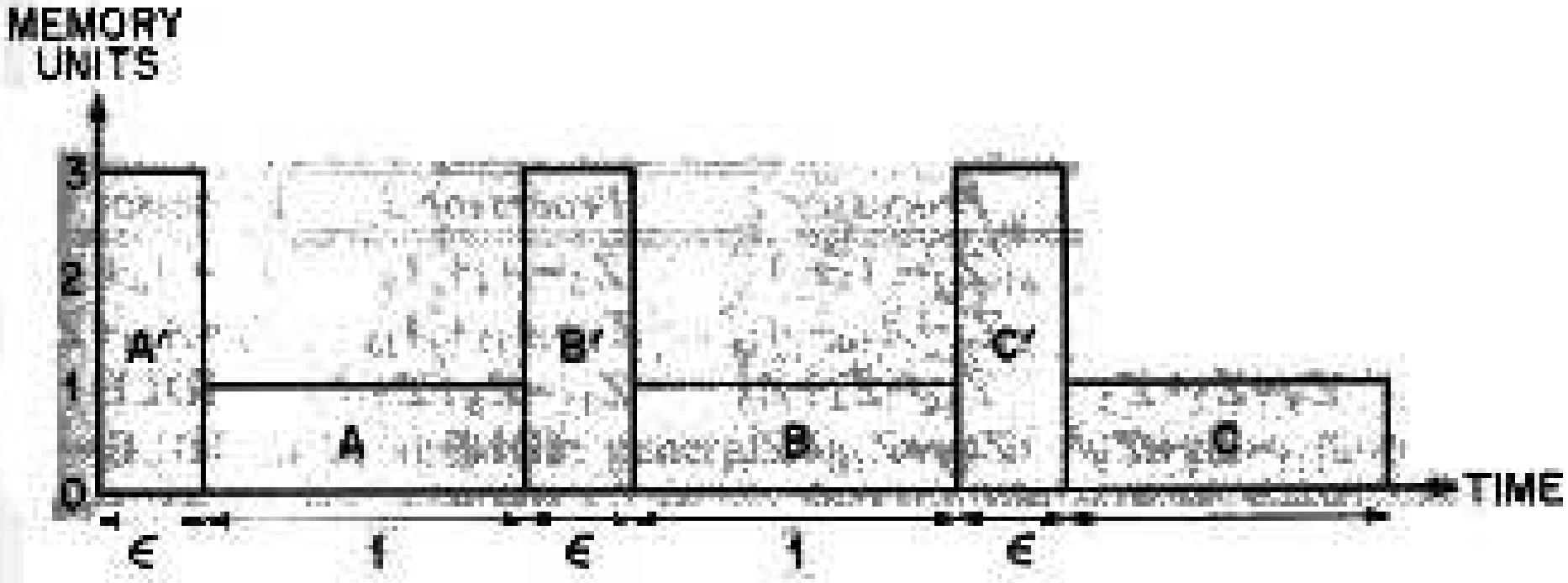


Figure 2.15.Schedule Two for Three Processors

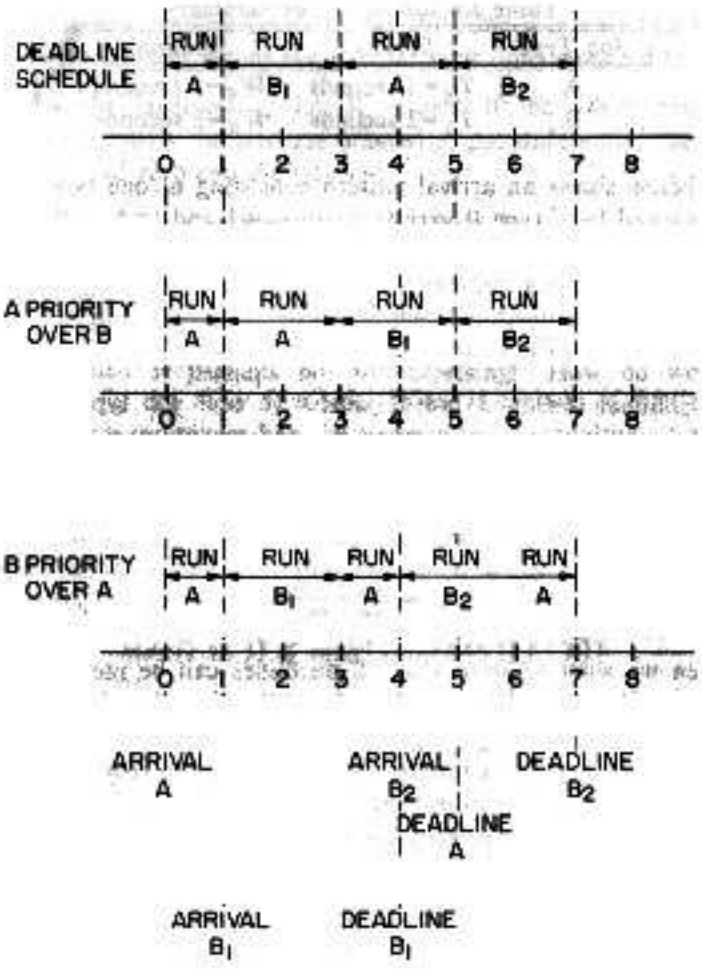


Figure 2.16.Illustrative Comparison of Deadline vs Static Priority Scheduling

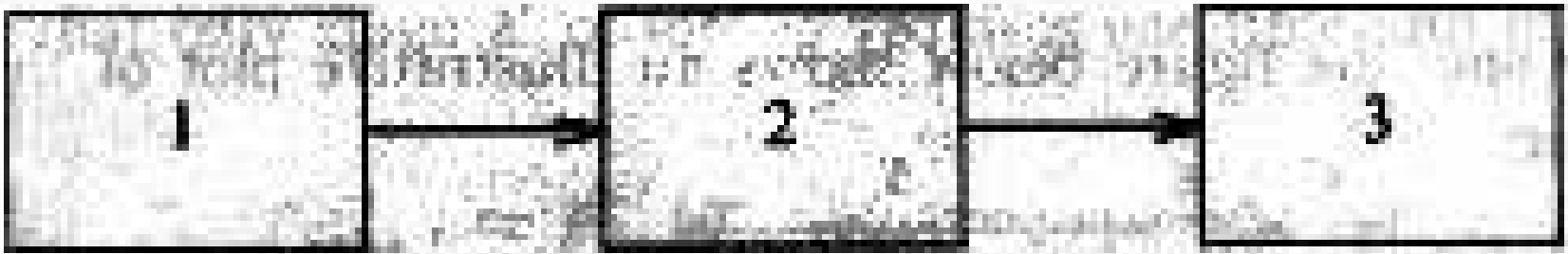


Figure 2.17.Three Stage Pipeline

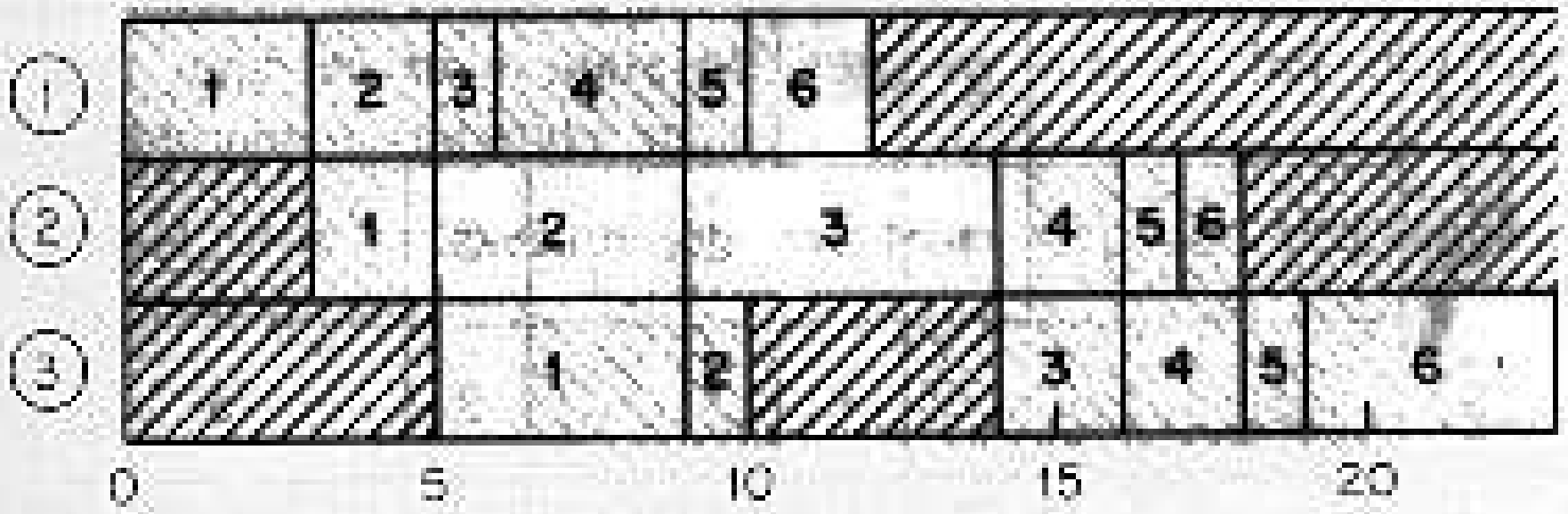


Figure 2.18.Three Stage Pipeline Schedule For Six Jobs

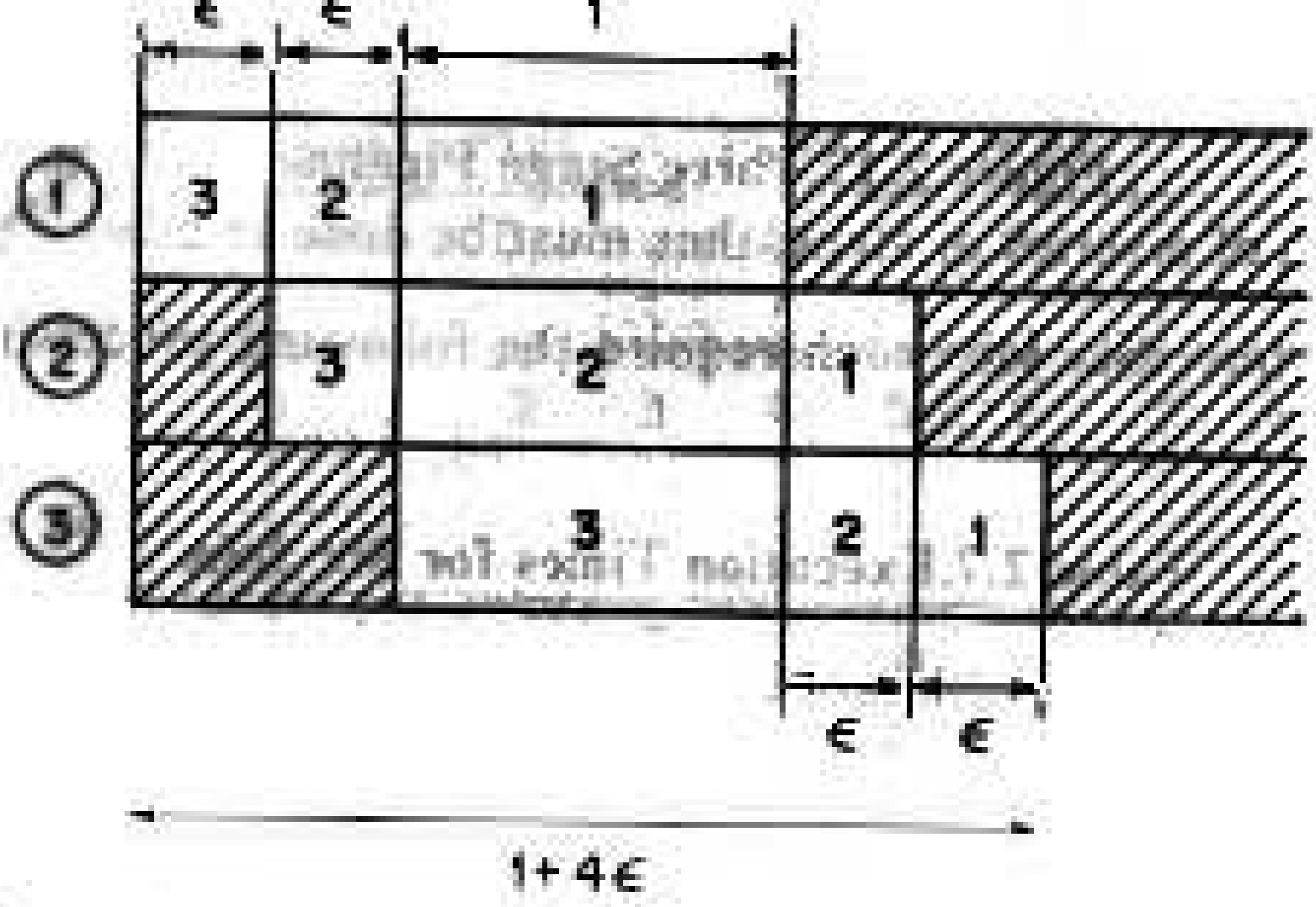


Figure 2.19.Schedule One Processor Activity (P = 3,N = 3)

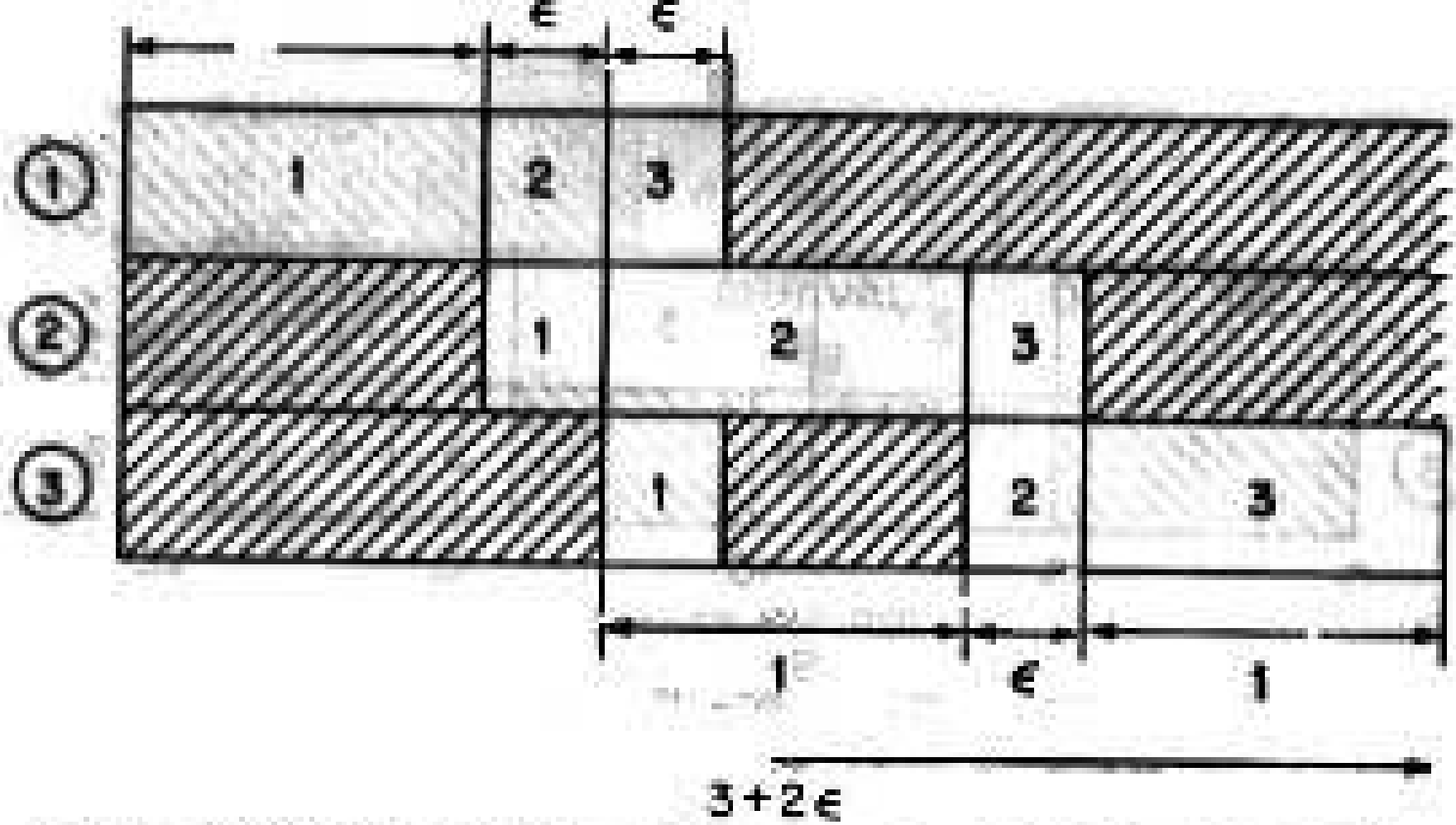


Figure 2.20.Schedule Two Processor Activity (P = 3,N = 3)

Figure 2.21.Two Stage Pipeline

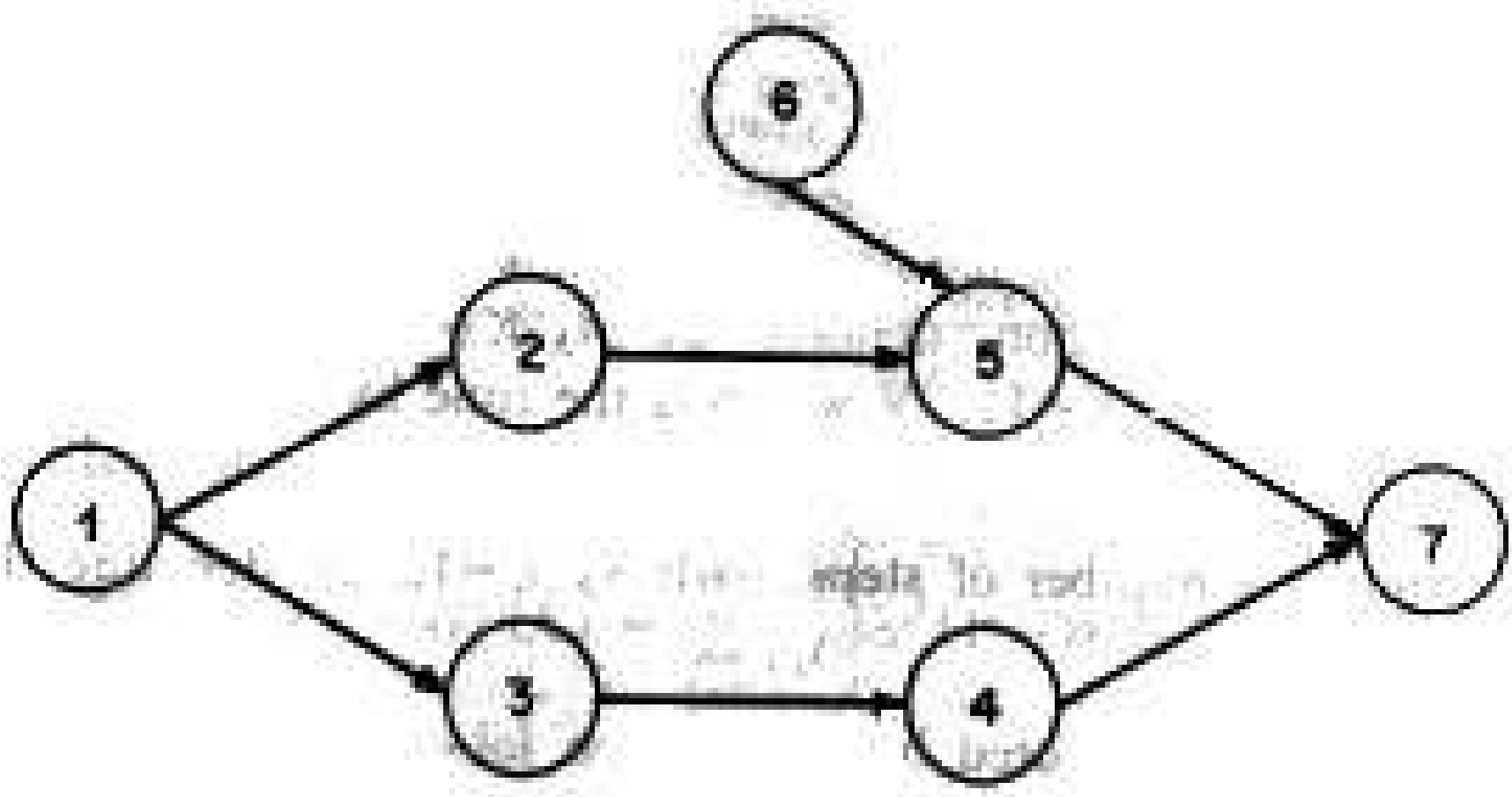
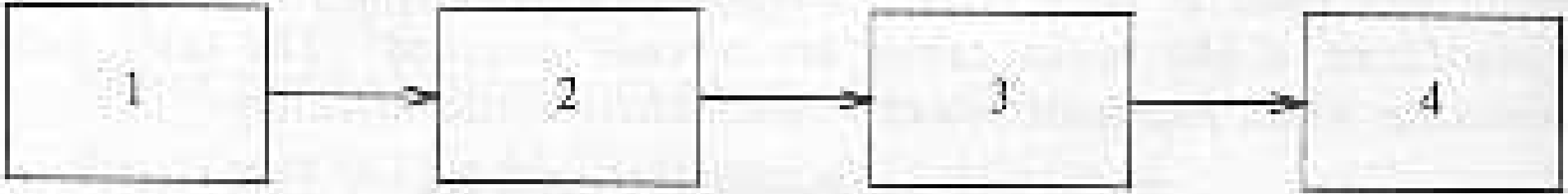


Figure 2.22.Precedence Constraints for Six Jobs



Four Stage Processor Pipeline