UNIX운영체제에서의 스케줄링 법칙과 큐잉응답 시간 분석

임 종 설[†]

요 약

본 논문은 UNIX 운영체제에서의 스케줄링 법칙에 대한 설명과 그에 따른 큐잉응답 시간을 분석한다. 큐잉응답시간은 UNIX내의 한 프로세서에 대한 조건부 응답시간의 평균값을 추정 분석함으로써 주어진다. 조건부 응답시간의 평균값은 일정한 CPU 시간을 필요로 하는 프로세서가 컴퓨터에 보내지는 시점에서 그 프로세서가 CPU 시간을 완료하고 되돌아오는 시점까지의 평균값이다. UNIX내의 스케줄링 법칙은 우선순위 서비스에 기초한다. 즉, 다음과 같은 단계의 스케줄링 법칙에 의하여 통제를 받는다. (i)시분할 사용량은 각요청에 대하여 필요한 CPU 시간을 완료할때까지 한개씩의 퀀텀(Quantum)을 할당함으로써 구해진다; (ii)퀀텀 할당을 받기 위하여 시스템 모드에서는 비선점(Nonpreemptive)교환, 사용자 모드에서는 선점(Preemptive)교환을 사용한다; (iii) 동일 우선순위내에서는 FCFS법칙을 사용한다; (iv)할당된 퀀텀을 완료한 프로세서는 우선순위에 맞는 실행 대기행열의 맨 뒤에 위치되어 CPU 서비스를 기다리거나, 디스크 대기행열에 위치되어 슬립(Sleep)상태로 들어간다. 이와같은 프로세서 스케줄링법칙은 사용자 모드에서 라운드로빈(Round-robin)효과를 창조한다. 본 논문에서는 라운드로빈 효과와 선점 교환을 사용하여 사용자모드에서의 프로세서 지연을 추정한다. 비선점 교환을 사용하여 시스템모드에서의 프로세서 지연을 추정한다. 또한 디스크 입출력에 의한 프로세서 지연도 고려한다. 조건부 응답시간의 평균은 지연시간의 합을 추정하여 구하여진다. 본 논문의 결과는 시스템시간을 필요로하는 프로세서가 우수한 응답시간을 가지며, 사용자시간을 필요로하는 프로세서의 지연만큼 시스템시간을 필요로하는 프로세서가 응답시간에서 혜택을 받는다는 것을 보여준다.

Scheduling Algorithms and Queueing Response Time Analysis of the UNIX Operating System

Jong Seul Lim[†]

ABSTRACT

This paper describes scheduling algorithms of the UNIX operating system and shows an analytical approach to approximate the average conditional response time for a process in the UNIX operating system. The average conditional response time is the average time between the submittal of a process requiring a certain amount of the CPU time and the completion of the process. The process scheduling algorithms in thr UNIX system are based on the priority service disciplines. That is, the behavior of a process is governed by the UNIX process schuduling algorithms that (i) the time-shared computer usage is obtained by allotting each request a quantum until it completes its required CPU time, (ii) the nonpreemptive switching in system mode and the preemptive switching in user mode are applied to determine the quantum, (iii) the first-come-first-serve discipline is applied within the same priority level, and (iv) after completing an allotted quantum the process is placed at the end of either the runnable queue corresponding to its priority or the disk queue where it sleeps. These process scheduling algorithms create the round-robin effect in user mode. Using the round-robin effect and the preemptive switching, we approximate a process delay in user mode. Using the nonpreemptive switching, we approximate a process delay in system mode. We also consider a process delay due to the disk input and output operations. The average conditional response time is then obtained by approximating the total process delay. The results show an excellent response time for the processes requiring system time at the expense of the processes requiring user time.

[†] 종신회원 : 선문대학교 전산학과 교수

논문접수 : 1994년 5월 20일, 심사완료 : 1994년 7월 29일

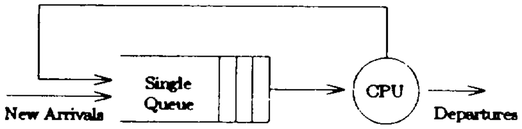
1. Introduction

This paper introduces the scheduling algorithms of the UNIX• operating system and shows an analytical approach to approximate the average conditional response time for a process in the UNIX operating system. The average conditional response time is the average time between the submittal of a process requiring x seconds of the CPU time (see CPU time in Section 4.1) and the completion of the process. The average conditional response time is a function of the CPU time - it is proportional to the CPU time. In the UNIX system, the CPU time consists of system time and user time (see Section 4.1). Therefore, the average conditional response time is a function of system time and user time. Also, the UNIX system has two classes of priorities - system priorities and user priorities and two types of modes - system mode and user mode (see Sections 3.1 & 4.1). For priorities, a system priority is higher than any user priority. The process scheduling algorithms in the UNIX system are based on the above priorities and modes. That is, a process is scheduled by the preemptive switching if a process is in user mode; by the nonpreemptive switching if the process is in system mode; by FCFS if the process is in the same priority level (see Section 3.3). From these process scheduling algorithms, we found the round-robin effect is created in user mode (see Section 3.4). Using the round-round effect and the preemptive switching, we approximate a process delay during its stay in user mode. Using the nonpreemptive switching, we approximate a process delay during its stay in system mode. We also considered a process delay during its stay for resources (e.g., block I/O). The average conditional response time is then obtained by approximating the total process delay (see Section 4.4).

The organization of this paper is as follows. Section 2 gives a brief overview of the well-known round-robin scheduling algorithm. Section 3 describes the process scheduling algorithms in the UNIX system. In Section 4, we introduce definitions, assumptions, and notation that are needed for our analyses. We also formulate a mathematical model of the average conditional response time and show approximate analyses. In Section 5, we present a case study for the exponential distribution. The case study gives some numerical results and discussions for the average conditional response time. Section 6 describes practical considerations of the derived results. By way of example, the average conditional response time of the UNIX system under an IBM-3081K machine is presented. Conclusions are given in Section 7.

2. The Round-Robin Scheduling Algorithm

In this section, we briefly describe the well-known round-robin scheduling algorithm. A newly arriving process joins the end of a single queue and waits until it finally reaches the CPU in a first-come-first-serve (FCFS) discipline. Upon reaching the CPU, the process seizes the CPU for the preassigned quantum. The process is then ejected from the CPU at the time it completes a preassigned quantum. Assuming that all quanta shrink to zero, the round-robin algorithm results in what is commonly known as the processor-sharing system. For the processor-sharing system, Kleinrock [3,4] and Sakata [8] derived the average response time of a process requiring x seconds of the CPU time as $T(x) = \frac{x}{1-\rho}$, where ρ = (average arrival rate of processes) × (average of the required CPU time x) = $\lambda\bar{x}$. If the process completes its required CPU time, it departs from the system and the CPU immediately executes the next process waiting at the head of the queue. If the process has not completed its required CPU time, it is cycled back to the end of the queue. The structure of this system is shown in Figure 1.



(Fig. 1) The Round-Robin System.

3. The UNIX Scheduling Algorithm

This section describes typical concepts as to how the UNIX process scheduling is done. Bach [1] discussed a detailed UNIX scheduling algorithm. Henry [2] showed the UNIX scheduler handles processes according to the prioritized round-robin scheduling algorithm. A brief discussion related to the UNIX scheduling algorithm is presented in Thompson [10]. Since source codes of the UNIX system are constantly modified, some of our descriptions in this section might not be appropriate for some cases.

3.1 Two Classes Of Priorities

In the UNIX system, the process scheduling algorithm is based on a priority service discipline using 128 priorities from 0 (high and good) to 127 (low and bad). There is one important dividing priority in these priorities, at 40. Priority 40 separates system priorities from user priorities (priority 40 is a user priority). A system priority is higher than any other user priority. A priority between 0 and 39 is assigned to a process that goes to sleep. A process goes to sleep when the resource (e.g., block I/O) it requests is unavailable and sleeps until the resource is obtained. When it wakes up, the process is placed at the end of the queue with the good priority (i.e., system priority).

3.2 Computation Of Priority

The priority of a process in system mode remains unchanged unless this process issues a system call (see system call in Section 4.1) and goes to sleep. The process goes to sleep and its priority is raised to a system priority only when the resource it requests is unavailable. A system priority is assigned according to the event for which the process is waiting. On the other hand, the priority of a process in user mode (i.e., user priority) changes dynamically, according to the recent CPU usage. The priority of a process in user mode decreases as it uses the CPU and increases as it waits for the CPU. The following example computes the priority of the process in user mode. The initial default user priority (usually 60) assigned to the process in user mode lasts only less than one second on the UNIX System V. It then receives a new value according to (3.1). A new user priority in (3.1) is computed every one second clock interrupt (e.g., every 60th clock tick)[1] or each time a process returns from a system call. The p_cpu_{now} in (3.1) is incremented by 1 in every 1/60 second clock tick while the process is executing on the CPU. Starting from zero, it can increase to a maximal value of 80. This zero value of p_cpu_{now} is assigned when a new process arrives, when a process is swapped in[2] , or when a process leaves the CPU for a long time (usually more than 7 seconds). The p_cpu_{now} is halved only at every 60th clock tick. Note that p_cpu_{now} becomes p_cpu_{before} at every 60th clock tick and that the smaller user priority in (3.1) implies the higher scheduling priority.

$$\text{user priority} = 60 + \begin{cases} p\_cpu_{now}/2 & \text{at the 60th clock tick} \\ p\_cpu_{now} & \text{otherwise,} \end{cases} \tag{3.1}$$

where p_cpu_{now} is computed by the following equation at every 60th clock tick just before the priority is computed by (3.1).

$$p\_cpu_{now} = \frac{1}{2}\{\text{Min}(80\,,\, p\_cpu_{before} + \text{the number of clock ticks while the process seizes the CPU})\}.$$

3.3 Scheduling With Four Disciplines

A newly arriving process usually gets a user priority 60 and joins at the end of the queue with priority 60. Also the process stays in user mode. Note a process may run either of two modes, namely user mode or system mode. It then accesses the CPU by FCFS. Upon seizing the CPU, the UNIX scheduler allots a quantum the process. A quantum is usually one second. If the process has not been interrupted during the full one second quantum, the process uses the CPU for the full one second and is then ejected to be placed at the end of the queue corresponding to its user priority. If the process

[1] In this example, one second has 60 clock ticks.

[2] For *swapping* policy, a process is swapped in from a secondary memory device. For a *demand paging* policy, a page is swapped in from a secondary memory device each time *page fault* occurs.

is interrupted by the higher priority process before a full one second quantum, the process is preempted (for this case, a quantum is less than one second). It is then placed at the end of the queue corresponding to its user priority. This discipline is *the preemptive switching in user mode*. When it gets the CPU later, its service continues from the point of preemption. On the other hand, if the process in a user priority issues a system call, the process transfers from user mode to system mode. In this case the priority of the process remains unchanged, but it is neither preempted by the process with the higher priority nor ejected by a quantum expiration. That is, the process completes the system call regardless of quantum and priority. This discipline is *the first type of the nonpreemptive switching in system mode*. During a system call, the process is put to sleep if the resource is unavailable. It then gets the system priority (high and good). Once the process with a system priority gets the CPU, it also be neither preempted by the process with higher priority nor ejected by a quantum expiration. This discipline is *the second type of the nonpreemptive switching in system mode*. Processes waiting on the same user or system priority are dispatched to the CPU, according to their arrival time on that priority. This discipline is *FCFS within a Priority Level*.

Upon completing the whole required CPU time, the process departs from the UNIX system and the CPU immediately executes the next process waiting at the head of the highest priority queue. If no process in the UNIX system, the CPU becomes idle.

3.4 Round-Robin Effect In User Mode

The UNIX process scheduling algorithm creates the round-robin effect in user mode (note that it does not create the round-robin effect in system mode). That is, the behavior of the processes in user mode do not exactly follow round-robin (see Section 2), but it resembles round-robin. To explain that, we consider the following three categories derived from the example in Section 3.2.

Category 1 : if a process in user mode has required an unavailable resource during one second quantum (i.e., if a process has to go to sleep), it will be placed at the end of a queue that is somewhere between 60th priority queue and 100th priority queue[3] when it returns to user mode from system mode.

Category 2 : a process in user mode has not required any resource and not been preempted during one second quantum, it will be placed at the end of a queue that is somewhere between 75th and 80th queue.

Category 3 : if a process in user mode has required an available resource or been preempted during one second quantum, it will be placed at the end of a queue that is somewhere between 60th and 80th queue.

From the above, we have the immediate result that a process in Category 1 will be on the average delayed longer than a process in Category 3. A process in Category 2 will be on the average delayed longer than a process in Category 3. In other words, a process that has required an unavailable resource or has completed a full one second quantum is delayed longer than a process that has not completed a full one second quantum. From this, we deduce the following:

- If a process has required an unavailable resource or has completed a full one second quantum, it is on the average delayed longer than other processes in the UNIX system;
- if a process has not completed a full one second quantum, it is on the average delayed shorter than other processes in the UNIX system.

We can then see that the delay in user mode is counterbalanced by the above two considerations so that the overall delay in user

[3] From (3.1), we can compute the new user priority ranging from 60 to 100.

mode may approach the same delay as in the round-robin scheduling algorithm. Moreover, as time progresses the priority of a process rises until it gets the CPU no matter which category it may belong to. Therefore, we believe the round-robin effect is created in user mode. This round-robin effect in user mode could be validated by the computer simulation. We plan to publish the computer simulation study on the round-robin effect in future. In this paper, we will use the round-robin effect in user mode without the exact proof.

4. An Analysis Of The UNIX Response Time

In this section, we introduce the following definitions, assumptions, and notation. Those will be used throughout the paper. We then formulate and analyze an approximate model.

4.1 Definitions

- CPU time : the required processing time of a process (CPU time consists of system time and user time).
- System mode : a UNIX mode where a process executes the UNIX kernel codes and accesses the system data segment.
- User mode : a UNIX mode where a process executes user programs and accesses the user data segment.
- System call : a call that is initiated by the processes in user mode and causes a transition from user mode to system mode.
- System priorities : UNIX priorities from 0 to 39 (high and good).
- User priorities : UNIX priorities from 40 to 127 (low and bad).
- System time : the time a process spends in system mode.
- User time : the time a process spends in user mode.
- Our tagged process : a process that we keep track of from its arrival (or submittal) to its departure (or completion) by attaching a tag.
- The average unfinished work U : the average CPU time required to empty all processes already in the UNIX system, assuming no new process enters the UNIX system (note that U equals the average waiting time of arriving processes in the queue by a FCFS discipline, but U is not equal to the average waiting time in the queue by the round-robin scheduling algorithm).
- The average modified unfinished work U′ : the average CPU time required to empty all processes already in the UNIX system, assuming no new process enters the UNIX system and assuming any process requiring the longer CPU time than our tagged process requires just the same CPU time as our tagged process.
- CPU factor : User time devided by CPU time.

4.2 Assumptions

A. The system has only one server (i.e., one CPU).

B. The arrival stream of processes and the requesting stream of system calls by processes are Poissonian at the average rates of λ and λ_s, respectively. The amount of the user time required by each arrival process and the amount of the system time required by each system call follow arbitrary distributions with the average of $\overline{x_u}$ and $\overline{t_s}$, respectively. The k^{th} moments of those are $\overline{x_u^k}$ and $\overline{t_s^k}$, respectively.

C. The context switching overhead between processes is negligible; all quanta in user mode are the same size and shrink to a negligibly small amount (presumably almost zero).

D. *The first type of the nonpreemptive switching in system mode* (see Section 3.3) does not happen. That is, we assume the requested resource is always unavailable. This assumption assures that a process in user mode has always a user priority and a process in system mode has always a system priority. Without this assumption, a process in system mode could have a user priority.

4.3 Notation

x_s the system time of our tagged process.

x_u the user time of our tagged process.

x the total CPU time of our tagged process $= x_s + x_u$.

p the CPU factor, x_u devided by x.

$T(x)$ The average response time of a process requiring x seconds of the CPU time (i.e., the average conditional response time).

U_s the average unfinished work for system time of processes in system priorities.

U'_u the average modified unfinished work for user time of processes found by our tagged process.

λ_s the average rate at which processes need the UNIX kernel (this rate results from system calls because a system call is generated each time processes need the UNIX kernel).

λ the average arrival rate of processes.

$\overline{t_s}$ the average system time required by each system call (i.e., the average duration of a system call).

$\overline{t_s^n}$ the n^{th} moment for the system time required by each system call.

$\overline{x_u}$ the average user time of all the processes entering the UNIX system.

$\overline{x_u}'$ our tagged process average delay by user time of new arrivals during its stay in the UNIX system.

$\overline{x_{x_u}^n}$ the n^{th} moment for the truncated remaining user time.

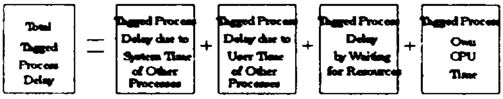
ρ_s the portion of time that the CPU executes processes in system mode (i.e., utilization for system time of processes $= \lambda_s \overline{t_s}$).

ρ_u the portion of time that the CPU executes processes in user mode (i.e., utilization for user time of processes $= \lambda \overline{x_u}$).

γ the expected portion of time that our tagged process is waiting for resources (e.g., block I/O) during its stay in the UNIX system.

4.4 Model Formulation

We begin by formulating our response time model from the viewpoint of our tagged process. Our tagged process is delayed on the average by the total system and user time of the processes waiting for the CPU during its stay at the UNIX system, by the waiting time for resources it needs, and by its own service time (i.e., by its required CPU time).



Total Tagged Process Delay
=
Tagged Process Delay due to System Time of Other Processes
+
Tagged Process Delay due to User Time of Other Processes
+
Tagged Process Delay by Waiting for Resources
+
Tagged Process Own CPU Time

Figure 2. The Total Tagged Process Delay.

Figure 2 above concisely illustrates the average conditional response time of our tagged process. As shown in Figure 2, the total tagged process delay equals the average conditional response time of our tagged process. Now, we can rewrite boxes 1-5 as $T(x)$, $T_1(x)$, $T_2(x)$, $T_3(x)$, and $T_4(x)$, respectively. Thus,

$$T(x) = T_1(x) + T_2(x) + T_3(x) + T_4(x)$$

Knowing that $T_3(x)$ and $T_4(x)$ are none other than $\gamma T(x)$ and x $(= x_s + x_u)$, respectively, we have

$$T(x) = T_1(x) + T_2(x) + \gamma T(x) + x \, . \qquad (4.1)$$

Since a system priority is higher than any user priority, we can consider the following two facts:

(1) Our tagged process obtains a user priority when it enters the UNIX system. It must wait for the CPU until all the processes (these do not include the sleeping processes) with system priorities complete their required system times. This delay is on the average the same as U_s.

(2) Also, if the processes enter the UNIX

system and issue new system calls during our tagged process stay T(x), each of those processes will get a system priority (by Assumption D in Section 4.2). Our tagged process must wait until new system calls generated by those processes complete. This delay is on the average $\lambda_s T(x)\overline{t_s}$. That is, $\lambda_s T(x)$ is the total number of new system calls during our tagged process stay and then each of the system call requires $\overline{t_s}$.

Thus, we have

$$T_1(x) = U_s + \lambda_s T(x)\overline{t_s}\,. \tag{4.2}$$

To derive U_s in (4.2), we observe that U_s consists of the following two parts (i.e., U_{s_1} and U_{s_2}) :

(1) U_{s_1} : Delay due to a process (if any) seizing the CPU with a system priority.

Because of the nonpreemptive switching in system mode, U_{s_1} is merely

$$U_{s_1} = \text{ARL} \times \rho_s = \frac{\overline{t_s^2}}{2\overline{t_s}}\rho_s \tag{4.3}$$

where ARL is the average residual life of system time of a system call and $\rho_s = \lambda_s\overline{t_s}$.

The idea in computing ARL is based on the assumption that the system time required by the process in a system priority is independently and identically distributed with an arbitrary distribution. This system time is renewed at the instant the CPU starts serving a new process in a system priority (refer to "Renewal Counting Process" in Parsen [Ch. 5, 5] and Ross [pg. 284, 6] for more details). Therefore, (4.3) becomes

$$U_{s_1} = \frac{\overline{t_s^2}}{2\overline{t_s}}\lambda_s\overline{t_s} = \frac{\lambda_s\overline{t_s^2}}{2} \tag{4.4}$$

(2) U_{s_2} : Delay due to the processes with a system priority that are found by our tagged process.

This delay is on the average the same as the average of the system time required by each system call ($\overline{t_s}$) times the average number of processes in system mode found by our tagged process (L_s). Note that L_s excludes the process (if any) seizing the CPU with a system priority. Thus,

$$U_{s_2} = \overline{t_s}L_s \tag{4.5}$$

Using Little's results (see Kleinrock [pg. 6, 4]), we can see that, on the average, $\lambda_s W_s$ processes will be in system priorities. Thus,

$$L_s = \lambda_s W_s\,, \tag{4.6}$$

where W_s is the average duration of a system call in a system priority.

Substituting (4.6) for L_s in (4.5),

$$U_{s_2} = \lambda_s\overline{t_s}W_s = \rho_s W_s\,. \tag{4.7}$$

Since we assume that the arrival stream of the system call is Poissonian and the duration of system call follows an arbitrary distribution, the average duration of a system call in a system priority is given by

$$W_s = \frac{\lambda_s\overline{t_s^2}}{2(1-\rho_s)}\,. \tag{4.8}$$

Refer to Kleinrock [pg. 16, 4] to see details for (4.8).

Substituting (4.8) into (4.7), we have

$$U_{s_2} = \frac{\rho_s\lambda_s\overline{t_s^2}}{2(1-\rho_s)}\,. \tag{4.9}$$

Since $U_s = U_{s_1} + U_{s_2}$, from (4.4) and (4.9) we obtain

$$U_s = \frac{\lambda_s\overline{t_s^2}}{2(1-\rho_s)}\,. \tag{4.10}$$

The equation (4.10) indicates that the average unfinished work U_s is the same as the average waiting time W_q by a FCFS discipline (refer to Conservation Laws in Kleinrock [pg. 113-118, 4] for more details). We can then rewrite (4.2) as

$$\begin{aligned} T_1(x) &= \frac{\lambda_s\overline{t_s^2}}{2(1-\rho_s)} + \lambda_s T(x)\overline{t_s} \\ &= \frac{\lambda_s\overline{t_s^2}}{2(1-\rho_s)} + \rho_s T(x)\,. \end{aligned} \tag{4.11}$$

4.5 Derivation Of $T_2(x)$

Our tagged process delay due to user time of other processes $T_2(x)$ is given by the average modified unfinished work U'_u. In addition, during our tagged process stay in the UNIX system, there will be on the average $\lambda T(x)$ new arrivals, each of which requires an average of $\overline{x_u}'$. Thus, we have

$$T_2(x) = U'_u + \lambda T(x)\overline{x_u}' . \qquad (4.12)$$

1) Derivation of U'_u

Recall that the behavior the processes in user mode resembles that of the round-robin scheduling algorithm and the overall delay in user mode approaches the same delay as in the round-robin scheduling algorithm (see Section 3.4). Therefore, we approximate U'_u by using the round-robin scheduling algorithm. Suppose that a process is already in the UNIX system and some of user time of that process has been completed by the time our tagged process arrives at the UNIX system. Now, we are interested in the remaining user time of that process. From the round-robin scheduling algorithm, it is evident that every process, having a remaining user time greater than x_u seconds at the instant our tagged process arrives at the UNIX system, contributes x_u seconds to U'_u. Also recall that the context switching overhead between processes is negligible (see Assumption C in Section 4.2). If the remaining user time of a process is more than x_u seconds, the view point of our tagged process changes. That is, our tagged process views that process as if it left the UNIX system upon completing x_u seconds of its remaining user time. Specifically, if a process requires a user time of s seconds and has received r seconds of the CPU service by the time our tagged process arrives, a remaining user time of the process is

$$s - r = y .$$

Then, its contribution to U'_u is

$$\begin{cases} y & y < x_u \\ x_u & y \geq x_u . \end{cases}$$

From the above, we obtain the truncated remaining user time density function that is the same as the residual life in the renewal counting process. Ross [pg. 44, 7] derived the cumulative density function (CDF) of the residual life as

$$\hat{F}(y) = \frac{\int_0^y [1 - F(t)]dt}{m} , \qquad (4.13)$$

where $F(t)$ is a cumulative distribution function of the required user time, and $m = \int_0^\infty [1 - F(t)]dt$. Thus, the truncated CDF of the remaining user time is

$$\hat{F}_{x_u}(y) = \begin{cases} \hat{F}(y) & y < x_u \\ 1 & y \geq x_u . \end{cases}$$

Now, the n^{th} moment of $\hat{F}_{x_u}(y)$ is derived by applying the Laplace-Stieltjes Transform (see Schrage [pg. 471, 9]) as

$$\overline{x^n_{x_u}} = \int_0^{x_u} y^n d\hat{F}(y) + x_u^n[1 - \hat{F}(x_u)] , \qquad (4.14)$$

where $\hat{F}(y)$ is given by (4.13).

Letting $n = 2$ in (4.14), we obtain the modified average unfinished work U'_u in (4.12) by using Pollacsek-Khintchine (P-K) formula (see Kleinrock [pg. 16, 4]) as follows.

$$U'_u = \frac{\lambda\overline{x^2_{x_u}}}{2(1 - \rho_u)}$$

$$= \frac{\lambda\{\int_0^{x_u} y^2\hat{F}(y) + x_u^2[1 - \hat{F}(x_u)]\}}{2(1 - \rho_u)} . \qquad (4.15)$$

2) Derivation of $\overline{x_u}'$

During our tagged process stay in the UNIX system, each of the newly arrival processes delay our tagged process an average of $\overline{x_u}'$ seconds.

Our tagged process, having visited the CPU n times so far, must have received an amount of user time equal to

$$Q_n = \sum_{i=1}^{n} q_i ,$$

where q_i is the i^{th} quantum for our tagged process and $1 \le n \le m$.

Note that Q_n is less than or equal to x_u and $Q_m = \sum_{i=1}^{m} q_i = x_u$.

Now, if the process that arrives at the UNIX system after our tagged process has arrived requires s of a user time, our tagged process with Q_n will be delayed by that process as

$$D(Q_n) = \begin{cases} s & \text{if } s < x_u - Q_n \\ x_u - Q_n & \text{if } s \ge x_u - Q_n \end{cases}. \qquad (4.16)$$

The equation (4.16) becomes possible by Assumption C in Section 4.2. This relationship and (4.16) lead to the following truncated distribution for our tagged process delay.

$$F_{x_u - Q_n}(s) = \begin{cases} F(s) & s < x_u - Q_n \\ 1 & s \ge x_u - Q_n \end{cases},$$

where F(s) is defined to be the required user time distribution of processes.

The first moment of the above $F_{x_u - Q_n}(s)$ is

$$\overline{x_{x_u - Q_n}} = \int_0^{x_u - Q_n} s\,dF(s) + (x_u - Q_n)[1 - F(x_u - Q_n)]. \qquad (4.17)$$

Then, using (4.17) we have

$$\overline{x_u}' = \sum_{n=1}^{m} \frac{q_n}{x_u} \overline{x_{x_u - Q_n}}. \qquad (4.18)$$

Since we assumed all quanta are the same size (see Assumption C in Section 4.2),

$$q_n = \begin{cases} q & \text{for } n=1, 2, \cdots, m\text{-}1 \\ q_m & \text{for } n=m, \end{cases}$$

where q is the equal quantum size and q_m is the last quantum size.

Then, we have $Q_n = nq$ (for n=1,2,...,m-1). Since we assumed all quanta shrink to negligibly small amount (see Assumption C), we have $q_m \approx q$ and $Q_m \approx mq$.

Thus, (4.18) becomes

$$\overline{x_u}' \approx \frac{q}{x_u} \sum_{n=1}^{m} \int_0^{x_u - nq} s\,dF(s) + (x_u - nq)[1 - F(x_u - nq)]. \qquad (4.19)$$

Now, we rewrite (4.12) as

$$T_2(x) = \frac{\lambda \overline{x^2_{x_u}}}{2(1 - \rho_u)} + \lambda T(x) \overline{x_u}', \qquad (4.20)$$

where $\overline{x^2_{x_u}}$ and $\overline{x_u}'$ are given by (4.14) and (4.19), respectively.

4.6 The Result Of the Analysis

The detailed procedures of deriving $T_1(x)$ and $T_2(x)$ were shown in the previous sections. Substituting (4.11) and (4.20) for $T_1(x)$ and $T_2(x)$ in (4.1), respectively, the equation (4.1) can be solved in terms of T(x) as

$$T(x) = \frac{\dfrac{\lambda_s \overline{t_s^2}}{2(1 - \rho_s)} + \dfrac{\lambda \overline{x^2_{x_u}}}{2(1 - \rho_u)} + x}{1 - (\rho_s + \lambda \overline{x_u}' + \gamma)}, \qquad (4.21)$$

where

λ_s, λ, and x were described in Section 4.3,

ρ_s is $\lambda_s \overline{t_s}$,

$\overline{x^2_{x_u}}$ is given by substituting 2 for n in (4.14),

ρ_u is $\lambda \overline{x_u}$, and

$\overline{x_u}'$ is given by (4.19).

Note the above T(x) is referred to as the average conditional response time because it is conditioned on the required CPU time of x seconds.

5. A Case Study

If the required time of each system call and of each arriving process follow the exponential, the evaluation of (4.21) becomes straightforward. The second moment of the exponential distribution function is given by

$$\overline{x^2} = 2(\overline{x})^2.$$

Thus, we have

$$\overline{t_s^2} = 2(\overline{t_s})^2, \qquad (5.1)$$

$$\overline{x_u}' \approx \overline{x_u}\left\{1+\frac{\overline{x_u}}{x_u}\left(e^{-\frac{x_u}{\overline{x_u}}}-1\right)\right\}, \text{ and} \qquad (5.2)$$

$$\overline{x^2_{\overline{x_u}}} = 2\overline{x_u}\left(\overline{x_u}-\overline{x_u}e^{-\frac{x_u}{\overline{x_u}}}-x_u e^{-\frac{x_u}{\overline{x_u}}}\right). \qquad (5.3)$$

See the Appendix for (5.2) and (5.3).

Substituting (5.1), (5.2), and (5.3) into (4.21),

$$T(x) \approx \frac{x+\frac{\rho_s}{1-\rho_s}\overline{t_s}+\frac{\rho_u}{1-\rho_u}\overline{x_u}\left(1-e^{-\frac{x_u}{\overline{x_u}}}-\frac{x_u}{\overline{x_u}}e^{-\frac{x_u}{\overline{x_u}}}\right)}{1-\left[\rho_s+\rho_u\left\{1+\frac{\overline{x_u}}{x_u}\left(e^{-\frac{x_u}{\overline{x_u}}}-1\right)\right\}+\gamma\right]} \qquad (5.4)$$

Note that $x = x_s + x_u$.

Now, we introduce the CPU factor, namely

$$p = \frac{x_u}{x_s + x_u} = \frac{x_u}{x}. \qquad (5.5)$$

Substituting $x_u = px$ from (5.5) into (5.4),

$$T(x) \approx \frac{x+\frac{\rho_s}{1-\rho_s}\overline{t_s}+\frac{\rho_u}{1-\rho_u}\overline{x_u}\left(1-e^{-\frac{px}{\overline{x_u}}}-\frac{px}{\overline{x_u}}e^{-\frac{px}{\overline{x_u}}}\right)}{1-\left[\rho_s+\rho_u\left\{1+\frac{\overline{x_u}}{x_u}\left(e^{-\frac{x_u}{\overline{x_u}}}-1\right)\right\}+\gamma\right]} \qquad (5.6)$$

From (5.6), we have

$$T(x) \approx \begin{cases} \dfrac{x_s+\frac{\rho_s}{1-\rho_s}\overline{t_s}}{1-(\rho_s+\gamma)} & \text{if } x_u \to 0 \\[2ex] \dfrac{x+\frac{\rho_s}{1-\rho_s}\overline{t_s}+\frac{\rho_u}{1-\rho_u}\overline{x_u}\left(1-e^{-\frac{px}{\overline{x_u}}}-\frac{px}{\overline{x_u}}e^{-\frac{px}{\overline{x_u}}}\right)}{1-(\rho_s+\rho_u+\gamma)} & \text{if } x_u \to \infty. \end{cases} \qquad (5.7)$$

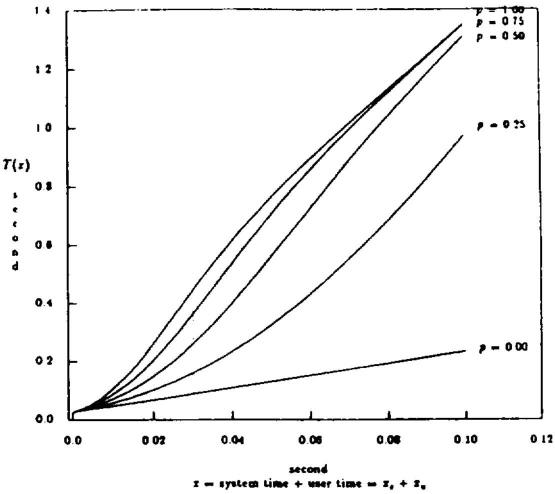
5.1 Discussions

In Figure 3, we display the response time T(x) as function of x for various p (i.e., p = 0, 0.25, 0.50, 0.75, 1.0). This figure happens to correspond to the case of exponential required time of each system call and of each arriving process with $\rho_s = 0.4$, $\rho_u = 0.4$, $\gamma = 0.1$, $\overline{t_s} = 0.001$, and $\overline{x_u} = 0.01$. From Figure 3 and (5.7), we note the following facts:

a. As the CPU time (= system time + user time) increases, the response time T(x) increases.

b. As the CPU factor p increases, the response time T(x) for a given CPU time x increases. In addition, the upper bound and the lower bound of T(x) are given when p = 0 and p = 1, respectively. This implies that the response time is greatly influenced, not only by the processing time (i.e., CPU time), but also by the CPU factor.

c. When p = 0 (i.e., when an arriving process does not require any user time) T(x) shows a linear increase. This linear increase is also shown by equation (5.7) for $x_u \to 0$. This implies that a process twice as long as some other will spend on the average twice as long in the UNIX system when $\overline{t_s}$ and x_u are negligibly small and $\rho_s + \gamma \ll 1$.



(Fig. 3) Response Time Curves for Various CPU factor p, $\rho_s = 0.4$, $\rho_u = 0.4$, $\gamma = 0.1$, $\overline{t_s} = 0.001$, and $\overline{x_u} = 0.01$.

d. Each arriving process will be delayed by at least

$$\frac{\frac{\rho_s}{1-\rho_s}\overline{t_w}}{1-(\rho_s+\gamma)} \qquad \text{if } x \rightarrow 0 . \qquad (5.8)$$

In other words, although the required CPU time of an arriving process approaches zero, the arriving process is still delayed by the amount of (5.8).

6. Practical Considerations

6.1 Considerations Of The Parameters

In practice, it is difficult to determine ρ_s, ρ_u, γ, $\overline{t_w}$, $\overline{x_u}$, and p in the derived response time equation (4.21). We will show how to determine those parameters in this section. The UNIX sar(1) (i.e., system activity reporter) gives %sys, %usr, %wio, and %idle.[4] For convenience, we let %sys + %usr + %wio denote the portion of time that the CPU is not purely idle. In contrast, %idle denotes the portion of time that the CPU is purely idle. Since ρ_s and ρ_u are utilizations for system time and user time, respectively, we have the immediate result such that

$$\%sys = \rho_s \text{ and } \%usr = \rho_u .$$

The γ was defined as the expected portion of time that our tagged process waits for resources (see Section 4.3). During T(x), our tagged process is waiting for resources each time the CPU is idle with some process waiting for resources. Thus, we can have the following conditional probability.

γ = pr{our tagged process is waiting for resources during its stay in the UNIX system}

= pr{the CPU is idle with some process waiting for resources given that our tagged process stays in the UNIX system} .

As long as our tagged process stays in the UNIX system, the CPU is either running in user mode, running in system mode, or idle with some process waiting for resources. In other words, the CPU can not be purely idle during the time our tagged process stays in the UNIX system.

Therefore, the above relationship becomes

γ = pr{the CPU is idle with some process waiting for resources / the CPU is not purely idle}

= pr{the CPU is idle with some process waiting for resources , the CPU is not purely idle}/pr{the CPU is not purely idle}

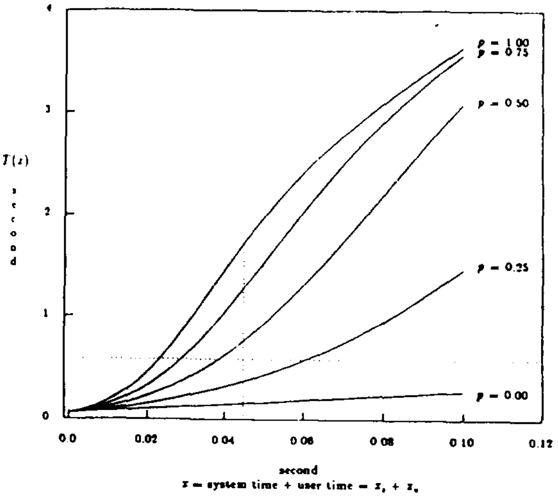
$$= \frac{\%wio}{\%sys + \%usr + \%wio}$$

$$= \frac{\%wio}{\rho_s + \rho_u + \%wio} .$$

Now, since the discussions in Section 5 are expressed only for the exponential distribution, we can pose a general question: will the same discussions be valid for the other distributions such as the Erlang-k, the uniform, and any arbitrary distribution ? To validate this, we need to examine cases using other types of distributions. Because of the complexity, we omit this examination in this paper and leave it for a future research.

6.2 Validation Of The Model

The following example demonstrates how to apply our results to the practical field. We drew sar(1) and acctcom(1) data generated from an



(Fig. 4) Response Time Curves for Various CPU factor p, ρ_s = 0.48, ρ_u = 0.44, %wio = 4%, $\overline{t_w} \approx 0$, $\overline{x_u}$ = 0.012.

[4] %sys, %usr, %wio, and %idle represent the portion of the time that the CPU runs in user mode, runs in system mode, is idle with some process waiting for block I/O, and otherwise is idle, respectively.

IBM-3081 machine for 15 minutes. This sampled data consisted of 6,306 processes. We then computed the average of system call duration, user time, and response time for the sampled 6,306 processes such that

$$\overline{t_s} \approx 0 \text{ sec}, \overline{x_u} = 0.012 \text{ sec}, \rho_s = 0.48, \rho_u = 0.44, \text{ and } \%\text{wio} = 0.04.$$

Substituting these values into (5.7), we obtain Figure 4. Then, we compared the sampled data with Figure 4. That is, we plotted the sampled response time from the UNIX OS running IBM-3081 machine onto Figure 4. We then found the sampled data appeared to lie more or less along the lines in Figure 4. Because it is straightforward, we do not display these validation plots in this paper.

7. Conclusions

We have analysed the response time of the processes under the UNIX operating system. We then presented a case study and an example of the practical application in the case of the exponential distribution. The results show the response time of a process requiring a certain CPU time is proportional to its CPU factor. This implies that the process requiring the larger portion of user time has the longer response time; the UNIX operating system is capable of providing an excellent response time for the processes requiring system time at the expense of the processes requiring user time.

References

[1] M. J. Bach, *The Design of The UNIX Operating System*, Prentice-Hall, 1986.

[2] G. J. Henry, "The Fair Share Scheduler," AT&T Bell Labs. Tech. J., Vol 63, No. 8, 1845-1857, 1984.

[3] L. Kleinrock, "Time-Shared Systems : A Theoretical Treatment," J. of A.C.M., Vol. 14, 242-261, 1967.

[4] L. Kleinrock, *Queueing Systems, Vol II: Computer Applications*, Wiley, 1976.

[5] E. Parzen, *Stochastic Processes*, Holden-Day, 1962.

[6] S. M. Ross, *Introduction to Probability Models*, Academic Press, Inc., 1985.

[7] S. M. Ross, *Applied Probability Models with Optimization Applications*, Univ. Microfilms International, Ann Arbor, 1970.

[8] M. Sakata, S. Noguchi, and J. Oizumi, "An Analysis of M/G/1 Queue under Round-Robin Scheduling," Operations Research, Vol. 19, 371-385, 1971.

[9] L. E. Schrage, "The Queue M/G/1 with Feedback to Lower .Priority Queues," Management Science, Vol. 13, No. 7, 1967.

[10] K. Thompson, "UNIX Time-Sharing System: UNIX Implementation," The Bell System Technical Journal, Vol. 57, No. 6, 1931-1946, 1978.

Appendix

Derivation of (5.2)

Since F(s) (i.e, the required user time distribution of processes) is of the exponential distribution, the integral part of (4.19) gives

$$\int_0^{x_u-nq} s\,dF(s) + (x_u - nq)[1 - F(x_u - nq)]$$

$$= \frac{1}{\overline{x_u}} \int_0^{x_u-nq} s\,e^{-\frac{s}{\overline{x_u}}}\,ds + (x_u - nq)e^{-\frac{x_u - nq}{\overline{x_u}}} . \quad \text{(A.1)}$$

Then, using the principle of *integration by parts*, (A.1) becomes

$$\overline{x_u} - \overline{x_u}\,e^{-\frac{x_u - nq}{\overline{x_u}}}$$

Thus, (4.19) gives

$$\overline{x_u}' \approx \frac{q}{x_u}\sum_{n=1}^{m}\left(\overline{x_u} - \overline{x_u}\,e^{-\frac{x_u - nq}{\overline{x_u}}}\right)$$

$$= \overline{x_u} - \overline{x_u}\frac{q}{x_u}e^{-\frac{x_u}{\overline{x_u}}}\sum_{n=1}^{m} e^{\frac{nq}{\overline{x_u}}} .$$

Since a quantum q shrinks to a negligibly small

amount (see Assumption C in Section 4.2), we have

$$\overline{x_u}' \approx \lim_{q \to 0}\left(\overline{x_u} - \overline{x_u}\frac{q}{x_u}e^{-\frac{x_u}{\overline{x_u}}}\sum_{n=1}^{m} e^{\frac{nq}{\overline{x_u}}}\right)$$

$$= \overline{x_u} + \frac{(\overline{x_u})^2}{x_u}\left(e^{-\frac{x_u}{\overline{x_u}}} - 1\right).$$

Derivation of (5.3)

Similarly, substituting 2 for n in (4.14) and using the principle of *integration by parts*, we can obtain the equation (5.3).

임 종 설

1979년 서울대학교 섬유공학과 졸업(학사)
1983년 University of Cincinnati 산업공학과(공학석사)
1986년 Polytechnic University Operations Research(공학박사)
1986년~91년 AT&T 벨연구소 책임연구원
1991년~93년 한국이동통신 책임연구원
1993년~현재 선문대학교 전산학과 조교수
관심분야 : Computer Network, 데이타통신