© British Computer Society 2003

RCPCP: A Ceiling-based Protocol for Multiple-disk Environments*

JUN WU[1], TEI-WEI KUO[2] AND CHIH-WEN HSUEH[1]

[1]*Department of Computer Science and Information Engineering, National Chung Cheng University, Chiayi, Taiwan 621, Republic of China*
[2]*Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan 106, Republic of China*
Email: junwu@cs.ccu.edu.tw, ktw@csie.ntu.edu.tw, chsueh@cs.ccu.edu.tw

Processes running in a multiple-disk environment may share non-preemptible resources on the processor and, at the same time, request services from disk subsystems. In this paper, we propose a methodology which is efficient and easy to implement for scheduling processes in multiple-disk environments. In other words, our methodology addresses the scheduling of real-time processes which may stop to wait for disk I/O without releasing any locked semaphores. Our proposed methodology is a variation of the well-known priority ceiling protocol to schedule processes running in multiple-disk environments. The capability of the proposed methodology is verified by a series of simulation experiments under different workloads of CPU-bound and I/O-bound processes in multiple-disk environments, for which we have some encouraging experimental results.

Received 26 July 2001; revised 13 September 2002

1. INTRODUCTION

Real-time process execution must be logically correct and also be done in a timely fashion. Processes whose deadlines can be violated but have hazardous results are called *hard real-time processes*; those which still contribute to the system after their deadlines expire are called *soft real-time processes*. Real-time process scheduling has been an active research topic in the past few decades and a lot of excellent works have been completed on the feasibility and schedulability issues of a real-time system. Researchers have proposed various optimal algorithms, such as the rate monotonic scheduling algorithm, the earliest deadline first algorithm and the least slack first algorithm, in different contexts [1, 2, 3, 4].

Recently, real-time process scheduling has been analyzed under different architectural assumptions [5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19]. In particular, Mok [4] proposed a monitor-based scheduling model for non-preemptible critical sections. The *priority ceiling protocol* (PCP) was proposed by Sha *et al.* [18] to handle process synchronization with different sizes of critical sections. In PCP, a process's resource request is blocked if its priority is no higher than the priority ceiling of any resource which has been locked by any other process. A process can inherit the highest priority of the processes it blocks. The definition of the priority ceiling of a resource is the highest priority of processes which may use the resource. Chen and Lin [20] proposed a variation of PCP for dynamic priority assignments, while PCP was originally designed for fixed-priority assignments. The stack resource policy (SRP), proposed by Baker [21], extends PCP for multiple-resource-unit synchronization and may adopt either dynamic or fixed-priority assignments; the maximum number of context switchings per process under SRP is no more than two. Liu and coworkers [13, 14, 19] proposed the imprecise computation model to trade the computation precision and system workloads. Kuo and Mok [10] considered adaptive system workloads to dynamically reconfigure the system. Han and Lin [7] explored real-time scheduling issues with precedence constraints. Kang *et al.* [9] and Natale and Stankovic [16] explored the scheduling problems with end-to-end deadlines. Bernat and Burns [5] and Koren and Shasha [12] considered the skippings of process execution in consecutive cycles. Mok and coworkers [15, 22, 23] and Han [24] explored the schedulability problems of real-time processes which may have different computation requirements in different periods.

Although a lot of researchers have done excellent jobs in real-time process scheduling, various simplified assumptions on resource synchronization still exist. In particular, little work has been done in scheduling processes under a more realistic system architecture. In this paper, we are interested in scheduling the CPU-bound and I/O-bound processes, which involve a lot of disk I/O and require stringent response requirements, such as information servers, multimedia systems, surveillance systems or soft-based control systems. Although these applications were running under a multiple-disk firm/soft real-time

*This paper is an extended version of a paper that appeared in the *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, 1999.

environment, they still need a well-designed concurrency control mechanism, because processes may share non-preemptible resources on the processor, such as shared variables and, at the same time, request services from the disk subsystem. If we directly apply the well-known PCP, it lowers the performance of the entire system by means of a very strict control mechanism to avoid priority inversion. This is why we relax the strict ceiling rules to obtain better system utilization and propose a PCP-based protocol called the reduced-ceiling priority ceiling protocol (RCPCP). Note that PCP was originally designed for a uniprocessor environment, where processes with locked resources are not allowed to stop and wait for I/O operations. We must emphasize that even if PCP allows a process to stop by itself, the holding of some semaphores by the process during a disk I/O period may create lengthy blocking times to other processes and seriously deteriorate the schedulability of the entire system.

Kim *et al.* [25] proposed a variation of PCP called the *ceiling adjustment scheme* (CAS) for the real-time systems with mixed hard and soft processes, which adjusts the ceiling of shared resources by using the *slack blocking time* of processes, where the slack blocking time is the maximum time in a period during which the process can be blocked without violating the schedulability. The design of CAS can improve the concurrency of the entire system with mixed hard and soft/firm real-time processes. It attempts to adjust the ceiling of resources to a level as low as possible while guaranteeing the predictability of hard deadline processes. Takada and Sakamura [26] proposed the *ceiling abort protocol* (CAP) to make PCP abortable. CAP assumes that the critical sections of processes are divided into two parts: the *abortable critical section* and the *unabortable critical section*. In CAP, a higher-priority process τ_h can abort a lower-priority process τ_l if it is in an abortable critical section and the priority ceiling of the critical section is lower than the priority of the process τ_h. The architectural assumptions of CAS [25] and the RCPCP to be proposed in this paper are different. We are more interested in real-time process scheduling of CPU-bound and I/O-bound processes. In particular, processes may share non-preemptible resources on the processor, such as shared variables and, at the same time, request services from the disk subsystem. We must also point out that CAP [26] cannot be directly applied to our problem because processes could request I/O services in an unabortable critical section. The capability of the proposed methodologies is verified by a series of simulation experiments, for which we have some encouraging experimental results. We must emphasize that while little work has been done in scheduling real-time processes involving I/O subsystems, the idea of resource synchronization proposed in this paper can be applied to other PCP-like protocols and other similar scheduling frameworks.

The major contributions of this paper are two-fold. (1) We explore the real-time resource allocation problem under a more realistic system architecture, where processes are sequences of CPU and I/O bursts. Scheduling protocols which target the maximization of the entire system utilization are proposed. (2) We explore the balance between the management of priority inversion and the utilization of system resources. We propose different PCP-based protocols with different restrictions on resource synchronization and deadlock-freeness guarantees. Note that the techniques proposed in this paper can be applied to many kinds of real-time applications which involve a lot of disk I/O and require stringent response requirements, such as information servers, multimedia applications, surveillance systems or soft-based control systems (e.g. Computer Numerical Control (CNC) which requires on-time CNC file access). We must emphasize that there is always a tradeoff between the management of priority inversion and the concurrency and utilization of the system. How to reach a balance in system utilization and deadline satisfaction is of paramount importance in the design of real-time system applications. This work targets the center of this research.

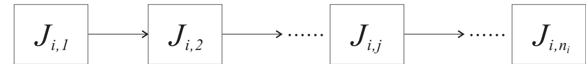

FIGURE 1. A real-time process in the multiple-disk environment.

The rest of this paper is organized as follows. Section 2 defines important terminologies. Section 3 illustrates the motivation of this research and proposes the basic protocol. In particular, we derive a very simple deadlock-detection mechanism for a variation of PCP with a high system utilization and a deadlock-free version. Section 4 provides simulation results for the proposed methodologies. Section 5 gives the conclusions of this paper.

2. SYSTEM MODEL AND DEFINITIONS

In this paper, we assume that a real-time system of periodic processes is running on a uniprocessor and may access multiple disks. For convenience, we assume that before a process accesses a resource, the process must lock the corresponding semaphore and, when a process terminates, it must unlock all of its semaphore locks (release them in the reverse order).

We are interested in the context of uniprocessor priority-driven preemptive scheduling and every process has a fixed priority. Each process τ_i is a sequence of jobs $J_{i,j}$, as shown in Figure 1. Let $P(\tau)$ denote the priority of process τ. Each process τ_i is associated with a period p_i and a sequence of the worst-case computation time requirements $c_{i,j}$ for each job $J_{i,j}$. Process execution consists of a cycle of CPU execution and I/O execution, where processes alternate back and forth between these two executions. Each process τ_i begins its execution with a CPU burst (by executing $J_{i,1}$ on the processor), followed by an I/O burst (by executing $J_{i,2}$ on one of the disks in the system), which is then followed by another CPU burst (by executing $J_{i,3}$) and so on. Eventually, the process τ_i ends with the last CPU burst (by executing J_{i,n_i} on the processor). We must emphasize that in exploring the tradeoff

between priority-inversion time and system utilization, a process may be blocked by more than one lower-priority processes. The maximum number and the average number of priority inversions experienced by each process will be shown theoretically and empirically, respectively, in the following sections. A process is a template of its instances and each instance will be instantiated for every request of the process. The requests of a *periodic process* τ_i will arrive regularly at the beginning of every period p_i. Each request of a periodic process τ_i should be completed no later than its deadline which is defined as its arrival time plus the relative deadline d_i of the process. Processes may share non-preemptible resources, such as shared variables, on the same processor.

We assume in this paper that critical sections are properly nested. In other words, locks are released in the reverse order from which they were obtained. Note that this is one of the assumptions of PCP in handling the priority-inversion problem. When there is no confusion, we will use terminologies 'process' and 'process instance' interchangeably.

3. THE REDUCED-CEILING PCP

3.1. Motivation

The well-known PCP [18] was originally designed to schedule a fixed set of periodic processes in a uniprocessor environment. Every resource in the system is guarded by a semaphore. The priority ceiling of a resource is defined as the highest priority of the processes which may lock the semaphore. The resource request of a process is granted if its priority is higher than the maximum-priority ceiling of any semaphores locked by other processes. Otherwise, the request is blocked. A process inherits the priority of any process which is blocked.

Although the idea of priority ceiling has been shown to be highly effective in managing the priority-inversion problem, it also introduces the unnecessary blocking of processes due to priority ceiling. Such blocking is obviously not a serious issue when processes only consume CPU cycles (as is the original assumption of PCP). However, when processes may stop and wait for I/O operations during their critical sections, the performance of the system may deteriorate very quickly. This is because the blocking time of the processes which are blocked by I/O-waiting processes can be very long and in the worst case the entire system may be forced to go idle.

The purpose of this work is to extend PCP in scheduling processes in multiple-disk environments. Our aim is to explore the tradeoff between a mechanism of priority-inversion control, such as PCP, and a better-utilization mechanism of CPU and I/O subsystems. If we directly apply the well-known PCP, it decreases the performance of the entire system by means of a very strict mechanism to manage priority inversion. This is why we relax the strict ceiling rules to obtain better system utilization and propose a PCP-based protocol called *reduced-ceiling priority ceiling protocol* (RCPCP). We will consider a process model in which process execution consists of a cycle of CPU execution and I/O execution, where processes will switch between these two executions.

3.2. The RCPCP

RCPCP is an extension of the well-known PCP [18] in scheduling processes which may request services from I/O subsystems. The rationale behind the design of the RCPCP is to lower the priority ceilings of semaphores held by processes which are waiting for the completion of I/O requests to improve the schedulability of the entire system.

The concept of priority ceiling was introduced by Sha *et al.* [18] to manage the priority-inversion problem in the system. Each semaphore S_i is associated with a priority ceiling PL_i which is equal to the highest priority of processes which may access S_i. Let $AccessSet_i$ and $LockedSet_i$ denote the sets of semaphores which may be accessed and are currently locked by process τ_i, respectively. We now present the definition of RCPCP.

1. The process which has the highest priority among all ready processes can execute on the processor. If a process does not attempt to lock any semaphore, the process can preempt the execution of any process with a lower priority, whether or not the priorities are assigned or inherited. (Priority inheritance will be defined later.)
2. When a process τ_i attempts to lock a semaphore S_j, the priority of τ_i must be higher than the priority ceilings of all semaphores currently locked by processes other than τ_i and S_j is unlocked; otherwise, the lock request is blocked. If τ_i is blocked because of some semaphore S^*, τ_i is said to be (directly) blocked by the process that locked S^*.
3. A process τ_i uses its assigned priority, unless it locks some semaphores and blocks higher-priority processes. If a process blocks a higher-priority process, it inherits the highest priority of the process blocked by τ_i. When a process unlocks a semaphore, it resumes the priority it had at the point of obtaining the lock on the semaphore. Moreover, the priority inheritance is transitive.
4. When a process τ_i is blocked on an I/O request, it is suspended for execution, the priority ceiling of each semaphore $S_j \in LockedSet_i$ is revised as follows:

$$PL_j = \min\Big\{original\ PL_j,\ \max_{S_k \in (AccessSet_i - LockedSet_i)} \{0, original\ PL_k\}\Big\},$$

   where 0 is the lowest priority level in the system. Note that PL_k is the original priority ceiling defined according to the priority ceiling definition. When process τ_i resumes after the I/O request is completed, the priority ceiling of each semaphore $S_j \in LockedSet_i$ is reset back to its original priority ceiling.

Our proposed algorithm could improve the schedulability of the entire system, because when a process τ_i stops to wait for disk I/O, the priority ceiling of each semaphore which is

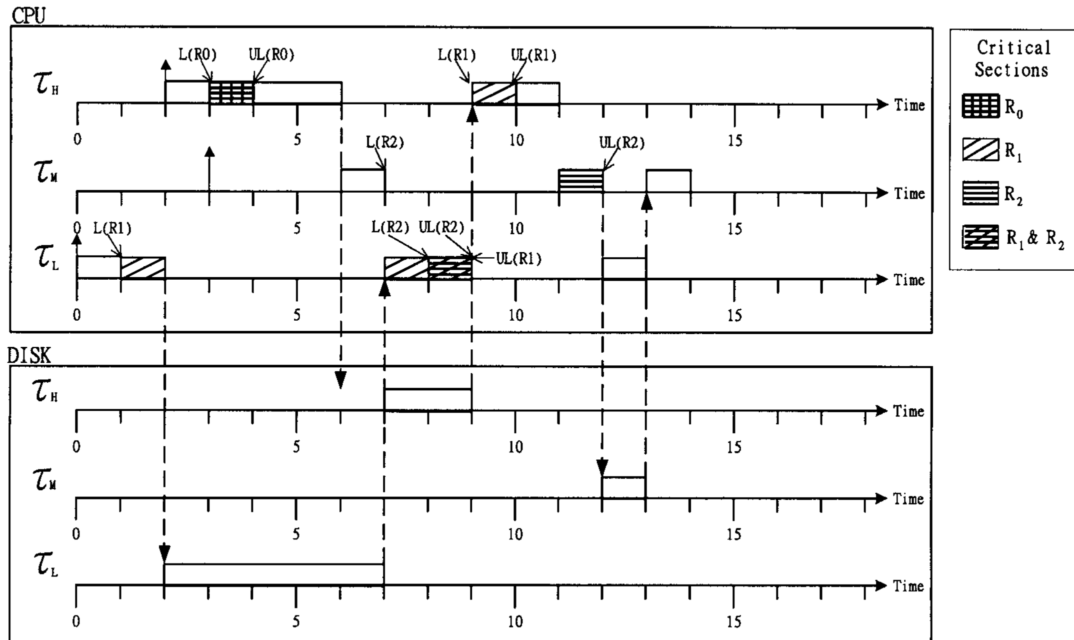

FIGURE 2. A RCPCP schedule.

locked by τ_i will be reduced. As a result, other processes will have less chance of being blocked by τ_i. In other words, the rationale behind RCPCP was to temporarily remove I/O-waiting processes from the competition of semaphores. Note that the possibility that a process τ_i may block other processes is because its locked semaphores are likely to decrease when more semaphores are locked by τ_i. When $(AccessSet_i - LockedSet_i) = \varnothing$, no semaphore locked by τ_i will block any other processes. We shall illustrate RCPCP with the following example.

EXAMPLE 1. (A RCPCP schedule) Suppose that there are three processes τ_H, τ_M, and τ_L in a uniprocessor system equipped with one disk I/O subsystem. Let the priorities of τ_H, τ_M and τ_L be 3, 2 and 1, respectively, where 3 is the highest and 1 is the lowest. Suppose that τ_H may access semaphores R_0 and R_1, τ_M may access semaphore R_2 and τ_L may access semaphores R_1 and R_2. According to the definition of priority ceiling, the priority ceilings PL_0 and PL_1 of R_0 and R_1 are both equal to 3 and the priority ceiling PL_2 of R_2 is equal to 2.

Figure 2 shows the execution of τ_H, τ_M and τ_L under RCPCP. At time 0, τ_L arrives and starts execution. τ_L locks R_1 successfully at time 1. At time 2, τ_L waits for a disk I/O request and PL_1 is set as $\min\{PL_1, PL_2\} = PL_2 = 2$. At time 2, τ_H arrives and starts execution. The lock request of τ_H on semaphore R_0 is successful at time 3 because the priority of τ_H is higher than the new priority ceiling of R_1 which is equal to $PL_2 = 2$. At time 3, τ_M arrives. Because the priority of τ_M is lower than the priority of τ_H, τ_H keeps running on the processor. At time 4, τ_H unlocks R_0. At time 6, τ_H starts waiting for a disk I/O request and τ_M starts execution. At time 7, the I/O request of τ_L is completed and PL_1 is reset back to 3. The lock request of τ_M on R_2 is blocked at time 7 because the priority of τ_M is no higher than $PL_1 = 3$. τ_L locks R_2 at time 8 and unlocks R_1 and R_2 at time 9. At time 9, the I/O request of τ_H is also completed and τ_H successfully locks R_1 because no semaphore is locked. τ_H unlocks R_1 at time 10 and completes its execution at time 11. τ_M resumes its execution at time 11 and successfully locks R_2. At time 12, τ_M unlocks R_2 and waits for an I/O request. τ_L resumes its execution on the processor at time 12 and finishes its execution at time 13. τ_M resumes its execution on the processor at time 13 and finishes its execution at time 14.

For comparison, Figure 3 shows the execution of τ_H, τ_M, and τ_L under PCP. Note that at time 3, the lock request of τ_H on semaphore R_0 is blocked under PCP because the priority ceiling PL_1 of R_1 held by the I/O-blocked process τ_L is not lowered. Furthermore, τ_M is also blocked by τ_L at time 4 when it tries to lock R_2. Such blocks issued by an I/O-blocked process keep the processor idle from time 4 to 7 when τ_L resumes from its I/O activity!

Example 1 shows the effectiveness of lowering the priority ceilings of semaphores held by I/O-blocked processes in improving the utilization of the system. However, we must point out that the reducing of priority ceilings during I/O operations may introduce deadlocks in the system and increase the number of priority inversions for processes. This is a price of a higher system utilization. In the following sections, we shall derive a bound for the number of priority inversions for processes and propose very simple deadlock detection and prevention mechanisms. Our experiments in Section 4 show that the number of priority inversions for processes is very small on average.

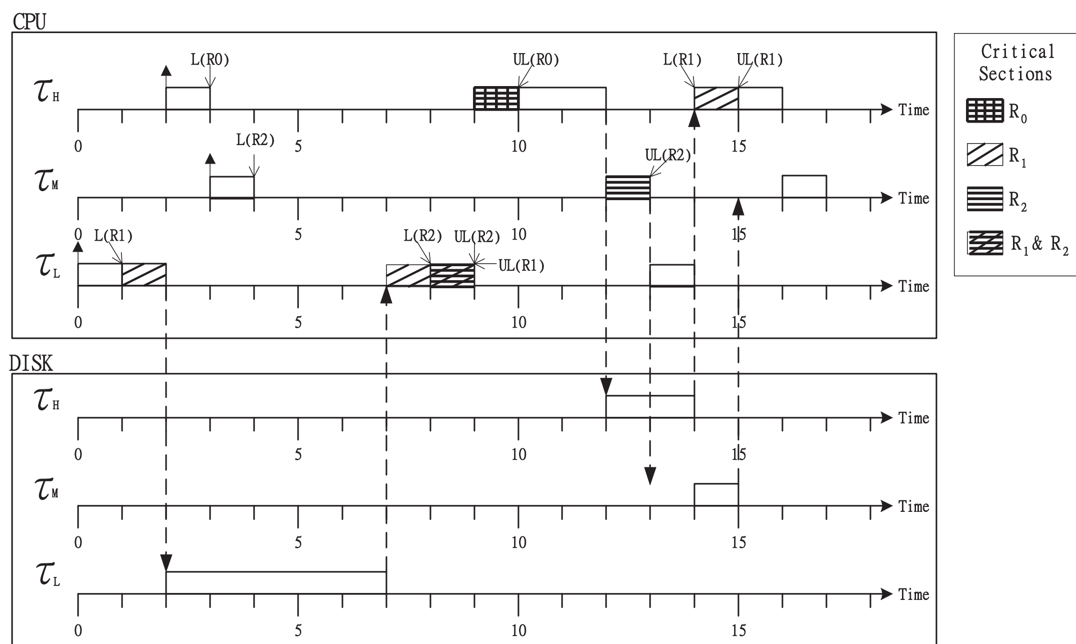

FIGURE 3. A PCP schedule.

3.3. Properties

The purpose of this section is to derive a bound for the number of priority inversions per process.

LEMMA 1. *A (higher-priority) process τ_H can be blocked by another (lower-priority) process τ_L only if τ_L is executing in a critical section which later blocks τ_H.*

Proof. According to the definition of the RCPCP, τ_L can block τ_H only if τ_L directly blocks τ_H because of a lock request, or τ_L inherits a priority higher than the priority of τ_H. In either case, τ_L must be in a critical section to block τ_H. Furthermore, if τ_L is not executing in a critical section, then τ_H can preempt τ_L because its priority must be no higher than the priority of τ_H. □

THEOREM 1. *The number of priority inversions for any real-time process under RCPCP is bounded by the number of semaphores in the system.*

Proof. Since Lemma 1 shows that a higher-priority process τ_H can be blocked by another lower-priority process τ_L only if τ_L is executing in a critical section which later blocks τ_H, no process will be blocked by more than n lower-priority processes, where n is the number of semaphores in the system. Note that because of priority inheritance, it is not possible for a middle-priority process to preempt a lower-priority process which already blocks a higher-priority process. □

Compared with PCP [18], the maximum number of priority inversions for a real-time process may be increased significantly. That is the price paid for a higher system utilization. In Section 4, we show the average number of priority inversions for a real-time process under RCPCP and PCP in our experiment results.

3.4. Deadlock detection

The purpose of this section is to derive a simple condition for deadlock detection which is caused by the lowering of priority ceilings of semaphores locked by I/O-waiting processes. We shall then provide a simple procedure to identify any possible deadlocks.

DEFINITION 1. [18] *Transitive blocking is said to occur if a process is blocked by another process which, in turn, is blocked by the other process instance.*

THEOREM 2. *A deadlock cycle exists iff there exist a process τ and a higher-priority process τ' in the deadlock cycle which satisfy the following conditions:*

1. *τ and τ' both hold a semaphore and wait for some other semaphores;*
2. *the priority of τ is no higher than the priority ceiling of some semaphore locked by τ';*
3. *the priority of τ' is no higher than the priority ceiling of some semaphore locked by τ.*

Proof. The only-if part of the proof can be done as follows. Suppose that there is a transitive blocking chain of processes $\tau'_1, \tau'_2, \ldots, \tau'_m$. Since Lemma 1 shows that each process τ'_i may block τ'_{i+1} if τ'_i is executing in a critical section which later blocks τ'_{i+1}, let τ'_i block τ'_{i+1} in terms of semaphore S'_i and τ'_m block τ'_1 in terms of semaphore S'_m. Since τ'_1 and τ'_2 both hold a semaphore and wait for another, there are two cases to consider: (1) τ'_1 locks S'_1 before τ'_2 locks S'_2; (2) τ'_2 locks S'_2 before τ'_1 locks S'_1. Note that (2) is not possible because τ'_2 holds S'_2 and waits for another semaphore. Such waiting makes the priority ceiling of S'_2 not less than the priority of τ'_2 and thus that of τ'_1, regardless of whether the priority ceiling of S'_2 is lowered. In other words, τ'_1 locks S'_1 before τ'_2 locks S'_2. This happens when τ'_2

locks S'_2 after τ'_1 is blocked on I/O request and the priority ceiling of S'_1 is lowered. (Assume that the semaphores to be locked by subsequent locking requests of τ'_1 have low-priority ceilings.) Since τ'_2 blocks τ'_3 in terms of semaphore S'_2, the priority ceiling of S'_2 must be no less than the priority of τ'_1. Since τ'_1 blocks τ'_2 in terms of semaphore S'_1, the priority ceiling of S'_1 must be no less than the priority of τ'_2. The only-if part of the proof is done by taking τ and τ' as τ'_1 and τ'_2, respectively.

The if part of the proof can be done as follows. Since τ holds a semaphore and waits for some other semaphores and the priority of τ is no higher than the priority ceiling of some semaphore locked by τ', τ is directly blocked by τ'. On the other hand, since τ' holds a semaphore and waits for some other semaphores and the priority of τ' is no higher than the priority ceiling of some semaphore locked by τ, τ' is directly blocked by τ. There is a deadlock between τ and τ'! □

Theorem 2 suggests a very simple deadlock detection algorithm. The condition for deadlock detection is as follows:

> *When there are two processes executing on the processor in a hold-and-wait situation and the priority ceiling of a semaphore locked by one process is no less than the priority of the other process and vice versa, then there is a deadlock.*

The condition can be checked whenever a lock request is blocked. When any two processes satisfy the deadlock-detection condition, the system should abort any of the two processes to break the deadlock.

3.5. Deadlock prevention

The proof of Theorem 2 provides an insight in deriving a deadlock-prevention scheme. The reason of having a deadlock in RCPCP schedules is that the lowering of the priority ceilings of the semaphores held by an I/O-waiting process τ lets other processes have a chance to lock semaphores which later result in a deadlock. Note that under PCP, such processes were unable to lock the semaphores which result in a deadlock until τ unlocks its semaphores. Unfortunately, such lowering of priority ceilings is necessary in the design of RCPCP in preventing processes which are waiting for I/O requests from blocking some other processes to execute on the processor and to lock semaphores.

For deadlock prevention, the grant of a lock request on a semaphore S_j by a process τ_i under RCPCP can be revised as follows.

Process τ_i may lock S_j if the following conditions are both satisfied.

1. The priority of τ_i must be higher than the revised priority ceilings of all semaphores currently locked by processes other than τ_i and S_j is unlocked.
2. The priority of τ_i must be higher than the original priority ceilings of all semaphores currently locked by processes other than τ_i, or the original priority ceiling of S_j is less than the priority of every I/O-waiting process.

Otherwise, the lock request is blocked. The second condition in the lock granting procedure is for deadlock prevention and the first condition is the same as that in the definition of RCPCP.

THEOREM 3. *RCPCP with the deadlock-prevention scheme is deadlock-free.*

Proof. The second condition to grant a lock consists of two parts: (1) the priority of τ_i must be higher than the original priority ceilings of all semaphores currently locked by processes other than τ_i; or (2) the original priority ceiling of S_j is less than the priority of every I/O-waiting process. Obviously, the satisfaction of the first item transforms RCPCP back to PCP which is already deadlock-free. Note that processes which satisfy the first item are considered to be preempting the processes which arrive earlier, because the later-arriving processes have a sufficiently high priority to lock semaphores. The second item means that no later-arriving process may block any process which arrives earlier. Therefore, there is no cycle of processes which hold and wait for locking semaphores. □

4. PERFORMANCE EVALUATION

The experiments described in this section are meant to assess the capability of RCPCP in scheduling processes which may request services from I/O subsystems. In order to understand and evaluate the performance of our scheduling approaches, we have implemented a simulation model of a multiple-disk environment which consists of a uniprocessor and several disks. We ran simulations with five scheduling approaches: *rate monotonic* (RM) [1] priority assignment without any concurrency control mechanism; SRP [21]; PCP [18]; and RCPCP, with and without deadlock-prevention schemes under different workloads of CPU-bound and I/O-bound processes in a multiple-disk environment. Note that we made some assumptions for SRP implementation. We assumed that semaphores were the only resource in the system and the number of each semaphore was 1. Eventually, the preemption level (which is a positive integer that is statically assigned to the process—when a process τ_i preempts τ_j, the preemption level of τ_i must be higher than τ_j) of each process τ_i was the same as its priority.

The rest of this section describes the performance metrics and the data set in our experiments and the simulation results for a system with one and two disks.

4.1. Data set and measurement

The primary performance metric of interest is the miss ratio of a process, referred to as *MissRatio*. The *MissRatio* of each process τ_i is the percentage of requests of process τ_i that violate its time constraints. Let num_i and $miss_i$ be the total number of process requests (excluding process instances whose absolute deadlines were over the simulation time) and the total number of deadline violations during an

experiment, respectively. The *MissRatio* of a process τ_i is calculated as $miss_i/num_i$. Another metric is the average number of priority inversions experienced by a process, referred to as *PINumber*. Let num_i and pin_i be the total number of process requests (excluding process instances whose absolute deadlines were over the simulation time) and the number of priority inversions experienced by a process during an experiment, respectively. *PINumber* is calculated as pin_i/num_i. The response time of a process is another useful metric for performance evaluation, referred to as *ResponseTime*. The *ResponseTime* of each process τ_i is a time interval between the process start time and finish time.

The test data sets were generated by a random number generator. The number of processes per process set was randomly chosen between 5 and 30. The number of jobs per process was randomly chosen between 1 and 20. The CPU utilization of a process set is equal to the sum of the ratio of the computation requirement and the period of every process in the set. In addition, the disk utilization of the system depended on CPU utilization and the CPU-bound degree of a process set, where the CPU-bound degree of a system reflected the ratio of the execution time of a process running a CPU burst. In addition, the I/O-bound degree of a process set was the execution time of a process running an I/O burst. Note that the sum of the CPU-bound degree and I/O-bound degree of a process set was equal to 1 and the CPU-bound degree of a system ranged from 0.1 to 0.9. Suppose the CPU-bound degree was X (i.e. the I/O-bound degree was $1 - X$) and the CPU utilization of the processor was $Y\%$, then the disk utilization of the system was $Y \times (1 - X)/X\%$. For example, when the CPU-bound degree was 0.2 and the CPU utilization of the processor was 20%, the disk utilization of the system was $20 \times (1 - 0.2)/0.2 = 80\%$.

When there were two disks in the system, the ratio of workload of *disk1* was referred to as f which ranged from 0.1 to 0.5. When $f = 0.2$, the workload of *disk2* was $1 - f = 0.8$. Suppose that the disk utilization of the system was 80% and $f = 0.3$, then the utilizations of *disk1* and *disk2* were $80\% \times 0.3 = 24\%$ and $80\% \times (1 - 0.3) = 56\%$. The experiments simulated process sets with a CPU utilization factor between 0% and 100%, while the corresponding disk utilization was also less than 200%, where the CPU utilization factor of a process set was equal to the sum of the ratio of the CPU computation requirement and the period of every process in the set. The maximum utilization of disks in our simulation model was $(number_of_disks_in_the_system) \times 100\%$.

In this paper, we are interested in real-time process scheduling under a more realistic system architecture, where processes are sequences of CPU and I/O bursts. (As a characteristic of ordinary I/O activities, we assumed that the deadline of a process may be longer than its period and the process will be killed when its deadline expires.) Each process set was simulated from time 0 to time 1,000,000. Over 20 process sets per utilization factor were tested in the multiple-disk environment and their results were averaged with a 95% confidence interval. The period of a process was randomly chosen in the range (100, 10,000).

TABLE 1. Parameters of simulation experiments.

Parameter	Value
Process number per set	(5, 30)
Job number per process	(1, 20)
CPU utilization factors	(0%, 100%)
f : ratio of workload of *disk1*	(0.1, 0.5)
CPU-bound degree	(0.1, 0.9)
Process period	(100, 10,000)
The number of semaphores locked per process	(1, 10)
The number of semaphores in the system	50
Simulation time	1,000,000

The priority assignment of processes follows the RM priority assignment [2]. The deadline of a process was a multiple (randomly chosen from 1 to 5) of its period. The CPU utilization factor of each process was no larger than 30% of the total CPU utilization factor of the process set. The number of semaphores locked by a process was between 1 and 10. The critical sections of a process in locking semaphores were evenly distributed and properly nested for the duration of the process. The total number of semaphores in the system was 50 to create sufficient conflict in resource access to observe the phenomenon of frequent priority inversion and deadline violations. The parameters are summarized in Table 1.

In the rest of this section, we present results from a series of simulation experiments over different workloads of CPU-bound and I/O-bound processes in multiple-disk environments with one and two disks. This paper aims at real-time process schedulings that involve a lot of disk I/O and require stringent response requirements (e.g. soft-based control systems, such as CNC control which requires on-time CNC file access).

4.2. Simulation results in a single-disk environment

The purpose of this section is to evaluate the performance of scheduling approaches including RM priority assignment without any concurrency control mechanism, SRP, PCP, RCPCP and RCPCP with a deadlock-detection scheme (RCPCP-DP) in single-disk environments. They were evaluated under different workloads of CPU-bound and I/O-bound processes.

Figures 4a and 4b show the miss ratios (with 95% confidence intervals) of the entire process set and the processes of the top 1/4 priority in the set, respectively, when the CPU-bound degree is 0.3. The CPU utilization of the process set ranged from 5% to 45% because the CPU-bound degree was 0.3 and the system involved a lot of disk activities. For example, when the CPU utilization of the process set was 30%, the disk utilization was 70%. In other words, Figures 4a and 4b show the miss ratios of I/O-bound processes. It was shown that SRP and PCP had similar performances and greatly outperformed RM. Note that SRP was similar to PCP because we made some assumptions for

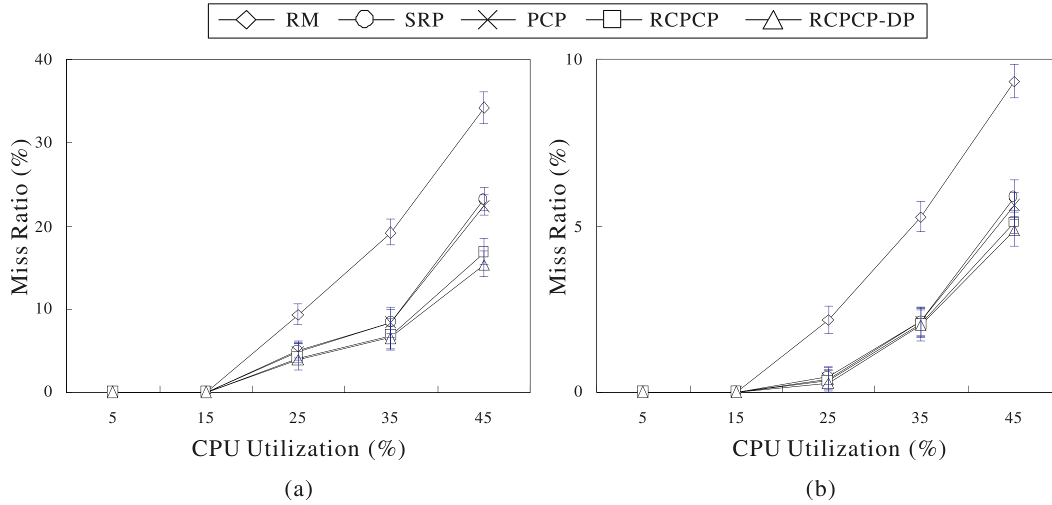

FIGURE 4. The miss ratio of processes when the CPU-bound degree is 0.3. Brackets indicate 95% confidence intervals. (a) The miss ratio of the entire process set; (b) the miss ratio of the processes of the top 1/4 priority.

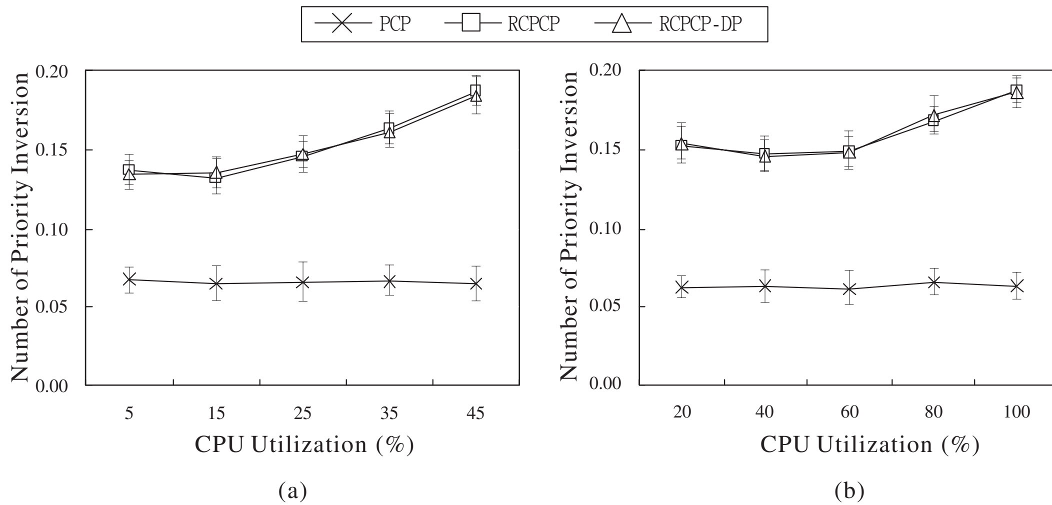

FIGURE 5. The average number of priority inversions of processes when the CPU-bound degrees are (a) 0.3 and (b) 0.7. Brackets indicate 95% confidence intervals.

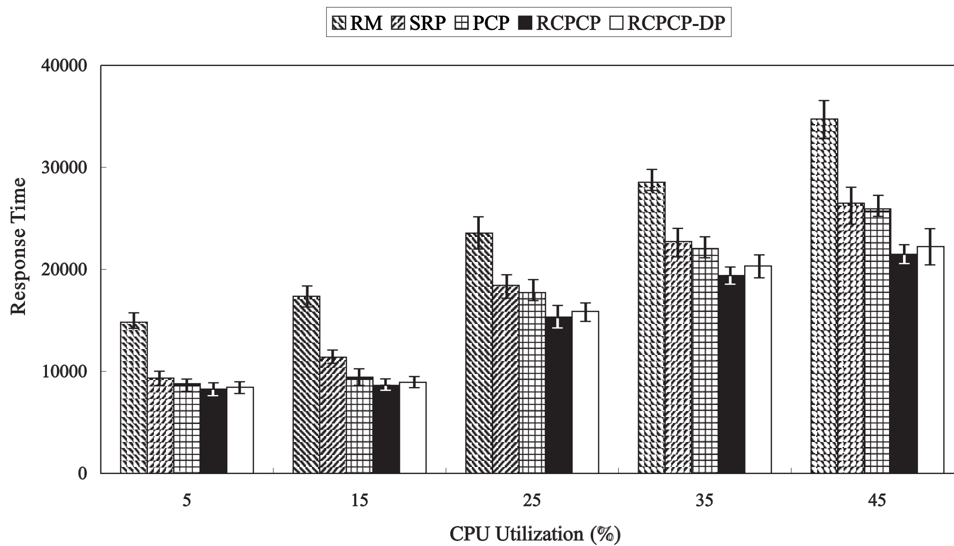

FIGURE 6. The average response time of processes when the CPU-bound degree is 0.3. Brackets indicate 95% confidence intervals.

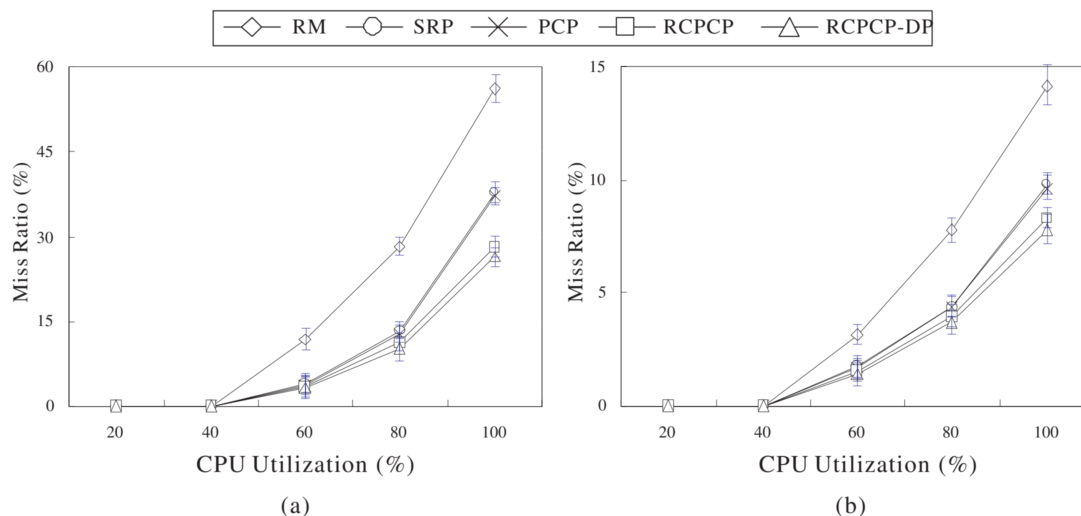

FIGURE 7. The miss ratio of processes when the CPU-bound degree is 0.7. Brackets indicate 95% confidence intervals. (a) The miss ratio of the entire process set; (b) the miss ratio of the processes of the top 1/4 priority.

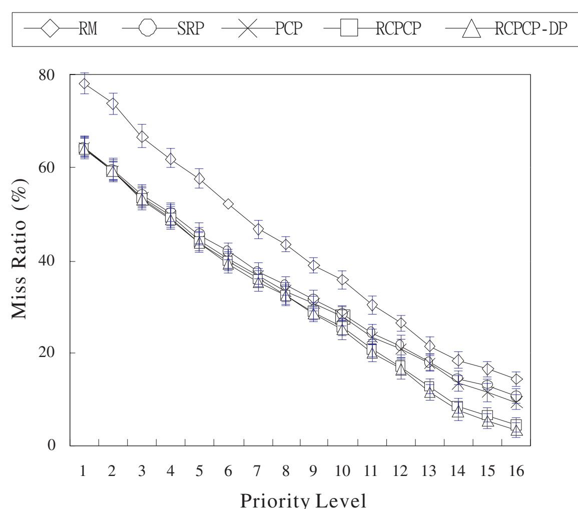

FIGURE 8. The average miss ratio of processes with different priority levels when there was one disk in the system. Brackets indicate 95% confidence intervals.

SRP implementation. We assumed that the semaphore was the only resource in the system and the number of each resource was 1. In addition, the preemption level of each process τ_i was the same as its priority. The only difference between SRP and PCP was that the processes under SRP approaches were blocked earlier than those under PCP.

Furthermore, Figure 4 shows that RCPCP and RCPCP-DP had similar performances and outperformed SRP, PCP and RM. It is obvious that the improvement of RCPCP and RCPCP-DP over other approaches became better when the CPU load of the processor was increasing. We can also observe that the higher the CPU utilization of the processor (i.e. the disk utilization of the processor), the better the improvement of RCPCP and RCPCP-DP over SRP, PCP and RM. As astute readers may point out, RCPCP-DP was better than RCPCP in scheduling processes. This was because processes scheduled by RCPCP, in fact, suffered from deadlocks in the experiments which caused some processes to miss their deadlines. However, the number of deadlocks was small. Although the design goal of RCPCP and RCPCP-DP was to lower the priority ceilings of semaphores held by I/O-waiting processes such that more processes (including lower-priority processes) may execute concurrently in the system, it was interesting to see that the miss ratios of the processes of the top 1/4 priority scheduled under RCPCP and RCPCP-DP were both decreasing. Thus, it is clear that our proposed approaches can improve the performance of the system which involved real-time CPU-bound and I/O-bound processes.

Figure 5a shows the average number of priority inversions (with 95% confidence intervals) of processes scheduled under RCPCP and RCPCP-DP was higher than that under PCP, when the CPU-bound degree was 0.3. We must point out that the number of priority inversions increased when more concurrency was obtained. Note that the maximum number of priority inversions for any real-time process under PCP is one [18]. Furthermore, for all of the results of the simulation experiments, the maximum numbers of priority inversions under PCP and RCPCP were 1 and 27, respectively. However, in average cases, the numbers of priority inversions under PCP and RCPCP were much lower (i.e. 0.068 and 0.137).

Figure 6 shows the average response time of processes (with 95% confidence intervals) when the CPU-bound degree was 0.3. Apparently, RCPCP and RCPCP-DP outperformed RM, SRP and PCP. It is also obvious that the improvement of RCPCP and RCPCP-DP over the other three approaches became better when the CPU load of the processor was heavier. When the CPU utilization was 45%, the average response time of processes under RCPCP and RCPCP-DP compared with that under PCP had 17% and 14% improvements.

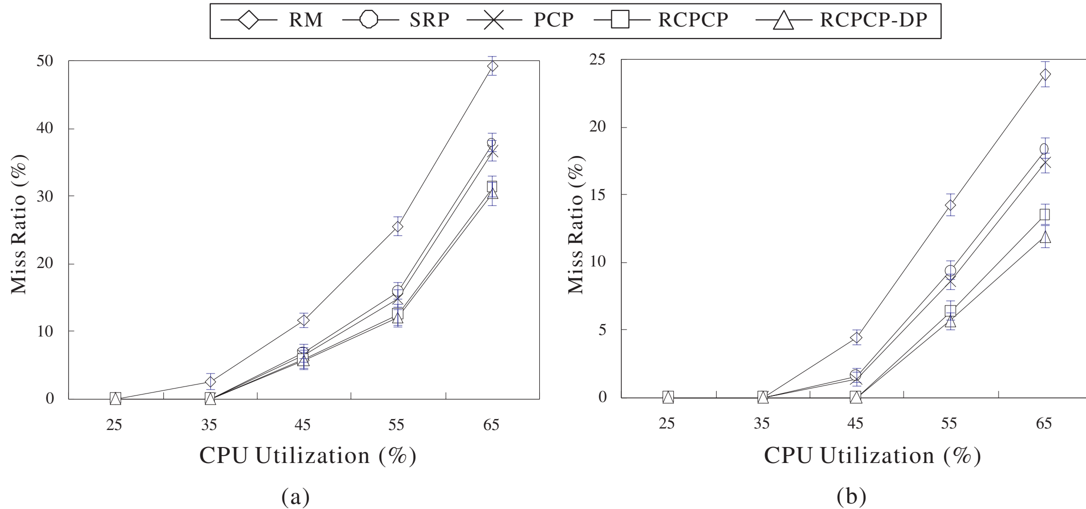

FIGURE 9. The miss ratio of processes when the CPU-bound degree is 0.3 and $f = 0.3$. Brackets indicate 95% confidence intervals. (a) The miss ratio of the entire process set; (b) the miss ratio of the processes of the top 1/4 priority.

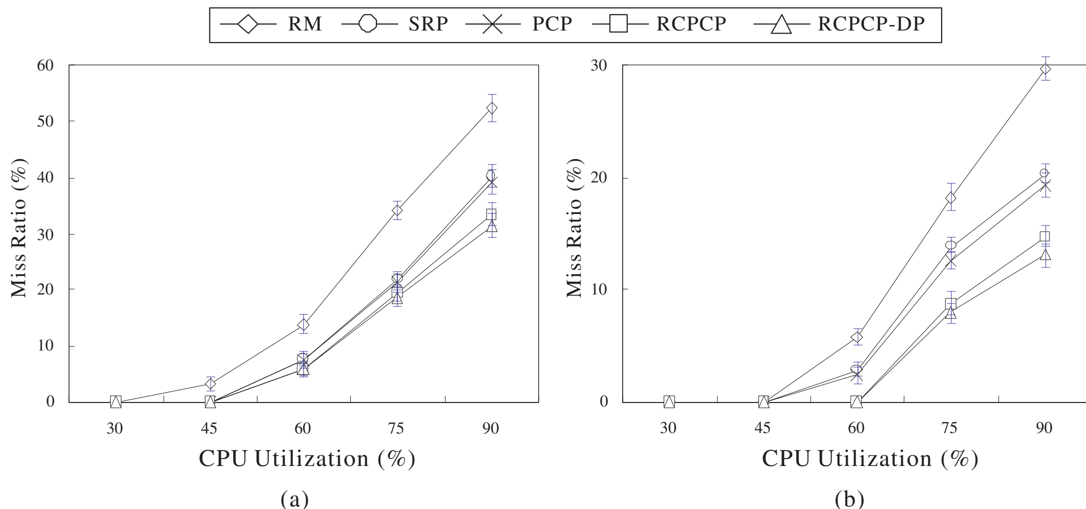

FIGURE 10. The miss ratio of processes when the CPU-bound degree is 0.3 and $f = 0.5$. Brackets indicate 95% confidence intervals. (a) The miss ratio of the entire process set; (b) the miss ratio of the processes of the top 1/4 priority.

Figure 7 shows the miss ratios of processes (with 95% confidence intervals), when the CPU-bound degree is 0.7. The system consisted of mainly CPU-bound processes. The CPU utilization of the process set ranged from 20% to 100%. We can also observe that the higher the CPU utilization of the processor (i.e. the lower the disk utilization of the processor), the better the improvement of RCPCP and RCPCP-DP over SRP, PCP and RM. The improvement of RCPCP and RCPCP-DP over PCP was not very significant because the disk utilization was low such that processes were only blocked infrequently by I/O activities. However, the improvement of the miss ratio of the processes of the top 1/4 priority was significant because they, on average, experienced less blocking time. Figure 5b shows the average number of priority inversions experienced by processes scheduled by PCP, RCPCP and RCPCP-DP, when the CPU-bound degree was 0.7. The average number of priority inversions experienced by processes scheduled by RCPCP and RCPCP-DP was higher than PCP. The results for CPU-bound degrees of 0.1 and 0.9 were not included because they were similar to those in Figures 4 and 7.

Figure 8 shows the average miss ratio of processes (with 95% confidence intervals) of different priority levels, where the priority level 16 is the highest and 1 is the lowest. As we can see, RCPCP and RCPCP-DP outperformed SRP, PCP and RM, especially for processes with higher-priority levels. This was because higher-priority processes could meet their deadlines more easily than those with lower priorities in priority-driven systems.

4.3. Simulation results in a multiple-disk environment

For a multiple-disk environment with two disks, we use f to denote the ratio of the workload of *disk1*, which ranged

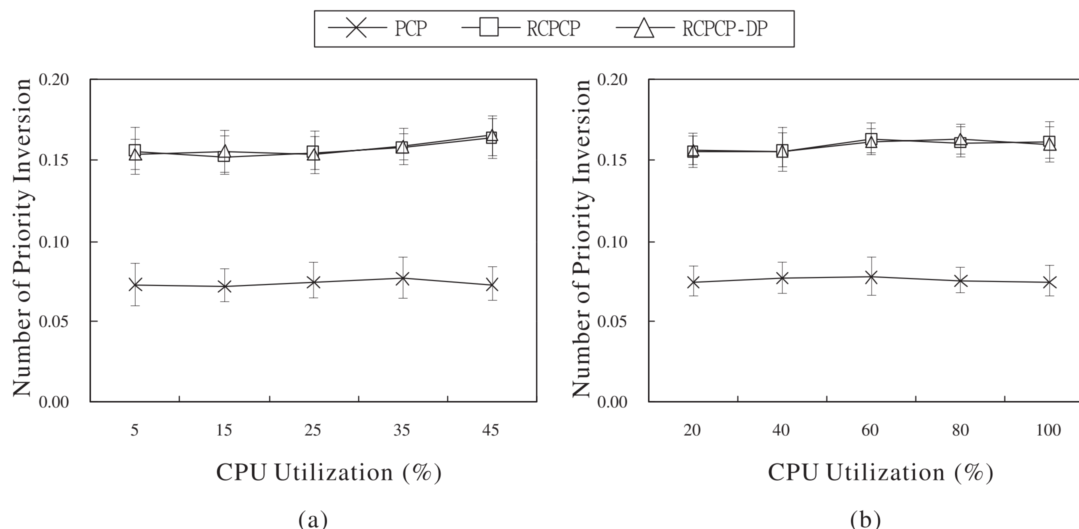

FIGURE 11. The average number of priority inversions of processes when the CPU-bound degree is 0.3: (a) $f = 0.3$ and (b) $f = 0.5$. Brackets indicate 95% confidence intervals.

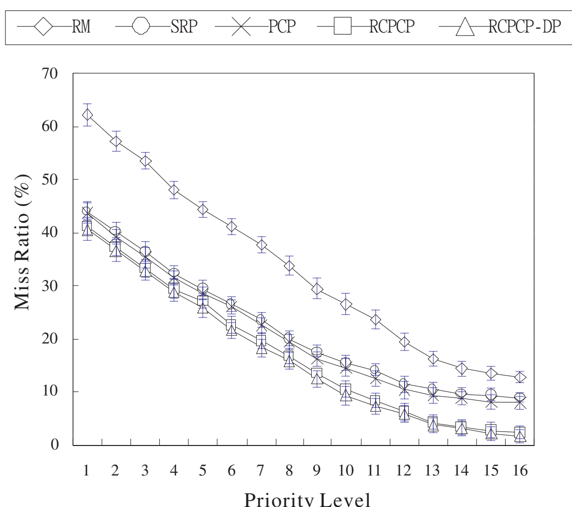

FIGURE 12. The average miss ratio of processes with different priority levels when there were two disks in the system. Brackets indicate 95% confidence intervals.

from 0.1 to 0.5. When $f = 0.5$, the ratios of workloads of *disk1* and *disk2* were balanced (50% and 50%). The ratios of workloads of *disk1* and *disk2* were unbalanced when $f = 0.1$ or 0.3. In the rest of our experiments, we present results on different workloads of CPU-bound and I/O-bound processes in a multiple-disk environment with two disks when $f = 0.3$ and 0.5. Note that when the number of disks was two, the maximum utilization of the disk was 200%.

Figures 9 and 10 show the miss ratios (with 95% confidence intervals) of processes when f was 0.3 and 0.5, respectively (the CPU-bound degree was 0.3). The system consisted of mainly I/O-bound processes. In Figures 9 and 10, the CPU utilization of the process set ranged from 25% to 65% and from 30% to 90%, respectively, because the CPU-bound degree was 0.3 and the system involved a lot of disk activities. For example, when the CPU utilization of the process set was 30%, the disk utilization was 70%. Because the workload ratio of *disk1* and *disk2* was 3:7 (i.e. $f = 0.3$), the disk utilization of *disk1* was 21% and that of *disk2* was 49%. In Figures 9 and 10, RCPCP and RCPCP-DP both significantly outperformed RM, SRP and PCP, especially for processes with higher priorities. For example, when the CPU utilization was 65% (as shown in Figure 9a), the *MissRatio* of processes under RCPCP had a 14% improvement compared with that under PCP. The only difference in parameter settings between Figures 9 and 10 was the value of f (i.e. 0.3 and 0.5). This reflected the workload ratio of *disk1* and *disk2*. When f was 0.5, the workloads of *disk1* and *disk2* were the same (i.e. even workloads). When f was less than 0.5, the workload of *disk2* was higher than that of *disk1* (i.e. uneven workload). In Figures 9 and 10, although the CPU-bound degree was the same (i.e. 0.3), the different values of f (i.e. 0.3 and 0.5) had impacts on the performance of simulation experiments. In particular, when CPU utilization was 60%, the *MissRatio* of the entire process set under RCPCP with an uneven workload of disks (i.e. $f = 0.3$ in Figure 9) was 21.3%. It was only 5.6% with an even workload of disks (i.e. $f = 0.5$ in Figure 10). Note that the workloads of *disk1* and *disk2* in Figure 9 were 42% and 98%, respectively. The workloads of *disk1* and *disk2* in Figure 10 were 70% and 70%, respectively. The high *MissRatio* of processes in Figure 9 was due to the heavy workload on *disk2*. The experimental results with CPU-bound degrees of 0.5 and 0.7 with $f = 0.3$ and 0.5 were not included because they were similar to those in Figures 9 and 10.

Figure 11 shows the average number of priority inversions when the CPU-bound degree was 0.3, $f = 0.3$ and 0.5. Figure 12 shows the average miss ratio of the different priority levels of processes in the set, where the priority level 16 is the highest and 1 is the lowest. Figure 13 shows the average response time of processes when the CPU-bound degree was 0.3 and $f = 0.5$. The results in

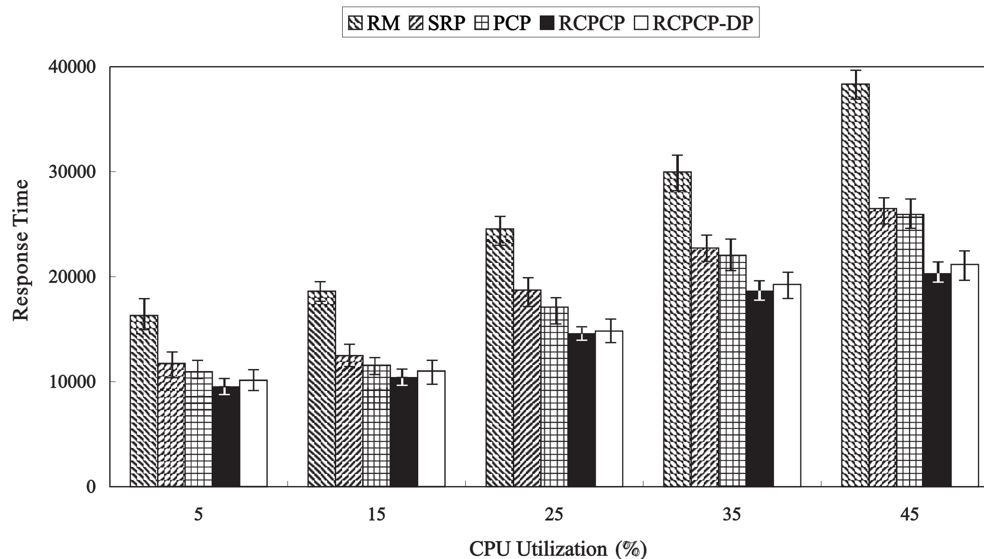

FIGURE 13. The average response time of processes when there were two disks in the system and the CPU-bound degree is 0.3 and $f = 0.5$. Brackets indicate 95% confidence intervals.

Figures 11–13 were similar to those in Figures 5, 8 and 6. The 95% confidence intervals of experimental results were also included in Figures 11–13.

Based on the simulation results for single- and multiple-disk environments, RCPCP and RCPCP-DP had similar performances and outperformed RM, SRP and PCP. The average number of priority inversions experienced by processes scheduled by RCPCP and RCPCP-DP was only a little higher than under PCP. The experimental results show that RCPCP had a good balance between the system utilization and the number of priority inversions. The miss ratio and average response time of processes decreased significantly under RCPCP and RCPCP-DP, compared with those under RM, SRP and PCP, especially when the system was heavily loaded.

5. CONCLUSIONS

This paper explores the real-time scheduling of processes which may share non-preemptible resources on the same processor and, at the same time, request services from other independent subsystems. In particular, we consider the scheduling of real-time CPU-bound and I/O-bound processes which may stop to wait for disk I/O without releasing any locked resources. A variation of PCP [18] called RCPCP with a better system utilization was proposed. We provide very simple deadlock detection and prevention mechanisms for RCPCP and prove its correctness. The capability of the proposed protocol is then verified by a series of simulation experiments. It was shown that when the system was moderately or heavily loaded, RCPCP and RCPCP-DP outperformed PCP. We must point out that the number of priority inversions increased when more concurrency was obtained.

Although the real-time scheduling problem has been analyzed under different architectural assumptions and frameworks, little work had been done in scheduling processes which may request services from independent I/O subsystems. For future research, we shall further explore real-time scheduling of CPU-bound and I/O-bound processes under different architecture platforms, such as those with multiple processors. More research in this area may prove to be very rewarding theoretically and practically.

REFERENCES

[1] Lehoczky, J. P., Sha, L. and Ding, Y. (1989) The rate monotonic scheduling algorithms—exact characterization and average behavior. In *Proc. 10th IEEE Real-Time Systems Symp.*, Santa Monica, CA, December, pp. 166–171. IEEE Computer Society Press.

[2] Liu, C. L. and Layland, J. W. (1973) Scheduling algorithms for multiprogramming in a hard real-time environment. *J. ACM*, **20**, 46–61.

[3] Leung, J. Y.-T. and Whitehead, J. (1982) On the complexity of a fixed-priority scheduling of periodic, real-time tasks. *Perform. Eval.*, **2**, 237–250.

[4] Mok, A. K. (1983) *Fundamental Design Problems for the Hard Real-time Environment*. PhD Dissertation, MIT, Cambridge, MA.

[5] Bernat, G. and Burns, A. (1997) Combining (n, m)-hard deadlines and priority scheduling. In *Proc. 18th IEEE Real-Time Systems Symp.*, San Francisco, CA, December 2–5, pp. 46–57. IEEE Computer Society Press.

[6] Burchard, A., Liebeherr, J., Oh, Y. and Son, S. H. (1995) New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Trans. Comput.*, **44**, 1429–1442.

[7] Han, C.-C. and Lin, K. J. (1992) Scheduling distance-constrained real-time tasks. In *Proc. 13th IEEE Real-Time Systems Symp.*, Phoenix, AZ, December 2–4, pp. 300–308. IEEE Computer Society Press.

[8] Han, C.-C. and Tyan, H.-Y. (1997) A better polynomial-time schedulability test for real-time fixed priority scheduling algorithms. In *Proc. 18th IEEE Real-Time Systems Symp.*, San Francisco, CA, December 2–5, pp. 36–44. IEEE Computer Society Press.

[9] Kang, D.-I., Gerber, R. and Saksena, M. (1997) Performance-based design of distributed real-time systems. In *Proc. 3rd IEEE Real-Time Technology and Applications Symp.*, Montreal, Canada, June 9–11, pp. 2–13. IEEE Computer Society Press.

[10] Kuo, T.-W. and Mok, A. K. (1997) Incremental reconfiguration and load adjustment in adaptive real-time systems. *IEEE Trans. Comput.*, **46**, 1313–1324.

[11] Kuo, T.-W., Liu, Y.-H. and Lin, K. J. (2000) Efficient on-line schedulability tests for the admission control of multimedia applications. In *Proc. IEEE Real-Time Technology and Applications Symp.*, Washington DC, May 31–June 2, pp. 4–13. IEEE Computer Society Press.

[12] Koren, G. and Shasha, D. (1995) Skip-over: algorithms and complexity for overloaded systems that allow skips. In *Proc. 16th IEEE Real-Time Systems Symp.*, Pisa, Italy, December 4–7, pp. 110–117. IEEE Computer Society Press.

[13] Liu, J. W. S., Lin, K. J. and Natarajan, S. (1987) Scheduling real-time periodic jobs using imprecise results. In *Proc. 8th IEEE Real-Time Systems Symp.*, San Jose, CA, December, pp. 252–260. IEEE Computer Society Press.

[14] Liu, J. W. S., Lin, K. J., Shih, W.-K., Yu, A. C., Chung, J. Y. and Zhao, W. (1991) Algorithms for scheduling imprecise computations. *Comput. Mag.*, **24(5)**, 58–68.

[15] Mok, A. K. and Chen, D. (1996) A multiframe model for real-time tasks. In *Proc. 17th IEEE Real-Time Systems Symp.*, Washington DC, December 4–6, pp. 22–31. IEEE Computer Society Press.

[16] Natale, M. D. and Stankovic, J. A. (1994) Dynamic end-to-end guarantees in distributed real-time systems. In *Proc. 15th IEEE Real-Time Systems Symp*, San Juan, Puerto Rico, December 7–9, pp. 216–227. IEEE Computer Society Press.

[17] Sha, L. (1992) *Distributed Real-time System Design Using Generalized Rate Monotonic Theory*. Lecture Note, Software Engineering Institute, CMU.

[18] Sha, L., Rajkumar, R. and Lehoczky, J. P. (1990) Priority inheritance protocols: an approach to real-time synchronization. *IEEE Trans. Comput.*, **39**, 1175–1184.

[19] Shih, W.-K. and Liu, J. W. S. (1992) On-line scheduling of imprecise computations to minimize error. In *Proc. 13th IEEE Real-Time Systems Symp.*, Phoenix, AZ, December, pp. 280–289. IEEE Computer Society Press.

[20] Chen, M.-I. and Lin, K.-J. (1989) *Dynamic Priority Ceilings: A Concurrency Control Protocol for Real-time Systems UIUCDCS-R-89-1511*, Department of Computer Science, University of Illinois at Urbana-Champaign.

[21] Baker, T. P. (1990) A stack-based resource allocation policy for real time processes. In *Proc. 11th IEEE Real-Time Systems Symp.*, Orlando, FL, December 4–7, pp. 191–200. IEEE Computer Society Press.

[22] Baruah, S. K., Chen, D. and Mok, A. K. (1999) Static-priority scheduling of multiframe tasks. In *Proc. 11th Euromicro Conference on Real-Time Systems*, York, UK, June 9–11, pp. 38–45. IEEE Computer Society Press.

[23] Mok, A. K. and Chen, D. (1997) A multiframe model for real-time tasks. *IEEE Trans. Software Eng,*, **23**, 635–645.

[24] Han, C.-C. (1998) A better polynomial-time schedulability test for multiframe tasks. In *Proc. 19th IEEE Real-Time Systems Symp.*, Madrid, Spain, December 2–4, pp. 104–113. IEEE Computer Society Press.

[25] Kim, J.-Y., Son, S. H. and Koh, K. (1997) The ceiling adjustment scheme for improving the concurrency of real-time systems with mixed workloads. In *Proc. 21st IEEE International Computer Software and Applications Conf.*, Washington DC, August 11–15, pp. 72–75. IEEE Computer Society Press.

[26] Takada, H. and Sakamura, K. (1994) Real-time synchronization protocols with abortable critical sections. In *Proc. 1st Int. Workshop on Real-time Computing Systems and Applications*, Seoul, Korea, December 14–16, pp. 48–52. IEEE Computer Society Press.