COP 4600 – Summer 2013

Introduction To Operating Systems

Multiprocessor Scheduling

Instructor : Dr. Mark Llewellyn
markl@cs.ucf.edu
HEC 236, 407-823-2790
http://www.cs.ucf.edu/courses/cop4600/sum2013

Department of Electrical Engineering and Computer Science
Computer Science Division
University of Central Florida

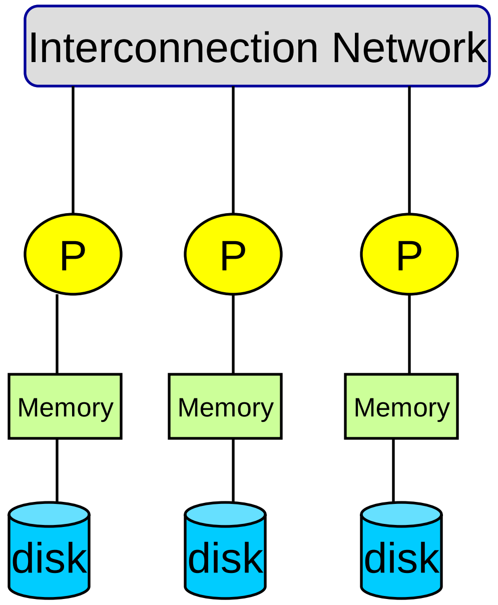
Scheduling In A Multiprocessor System

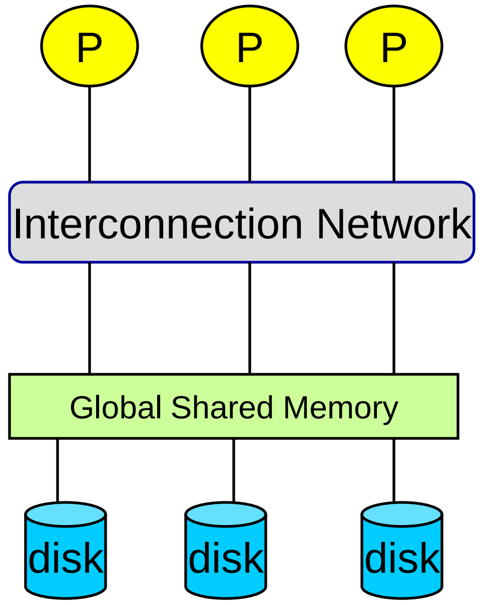
- When a computer system contains more than one processor, several new issues are introduced into the design of scheduling protocols.
- Load sharing becomes an issue for the scheduler.
- As with uniprocessor scheduling protocols, there is no one best protocol that will suffice for all situations.
- For the most part we will only be concerned with homogeneous systems, in which the processors are identical in terms of their functionality.

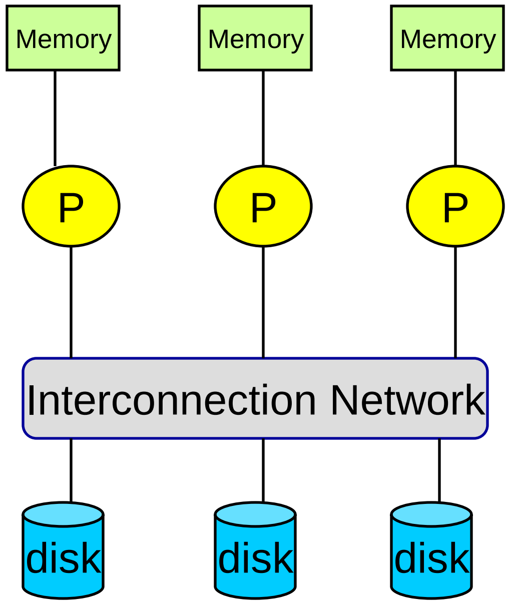
Classifications of Multiprocessor Systems

- Loosely coupled, distributed multiprocessors, or clusters
  - Fairly autonomous systems.
  - Each processor has its own memory and I/O channels
- Functionally specialized processors
  - Typically, specialized processors are controlled by a master general-purpose processor and provide services to it. An example would be an I/O processor.
- Tightly coupled multiprocessing
  - Consists of a set of processors that share a common main memory and are under the integrated control of an operating system.
  - We'll be most concerned with this group.

Multiprocessor Systems







Shared Nothing

(Loosely Coupled)

Shared Memory

(Tightly Coupled)

Shared Disk

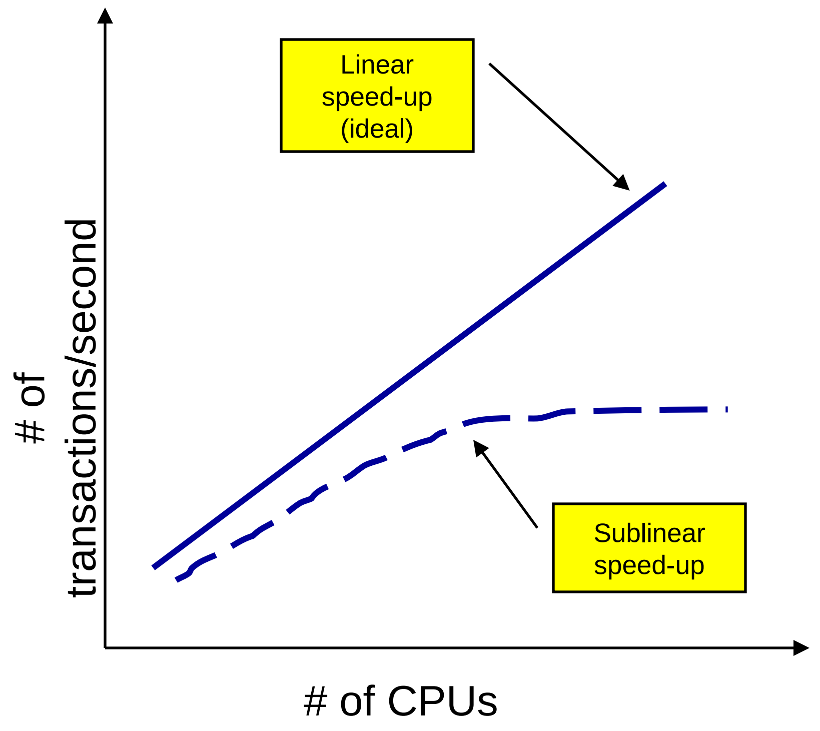
Multiprocessor Systems

- The basic problem with the shared-memory and shared-disk architectures is interference.
- As more CPUs are added, existing CPUs are slowed down because of the increased contention for memory accesses and network bandwidth.
- It has been shown that:
  - An average of 1% slowdown per additional CPU limits the maximum speed-up to a factor of 37.
  - Adding additional CPUs actually slows down the system.
  - A system with 1000 CPUs is only 4% as effective as a single CPU.
- These observations motivated the development of the shared-nothing architectures for parallel systems. These systems are particularly useful for database systems.

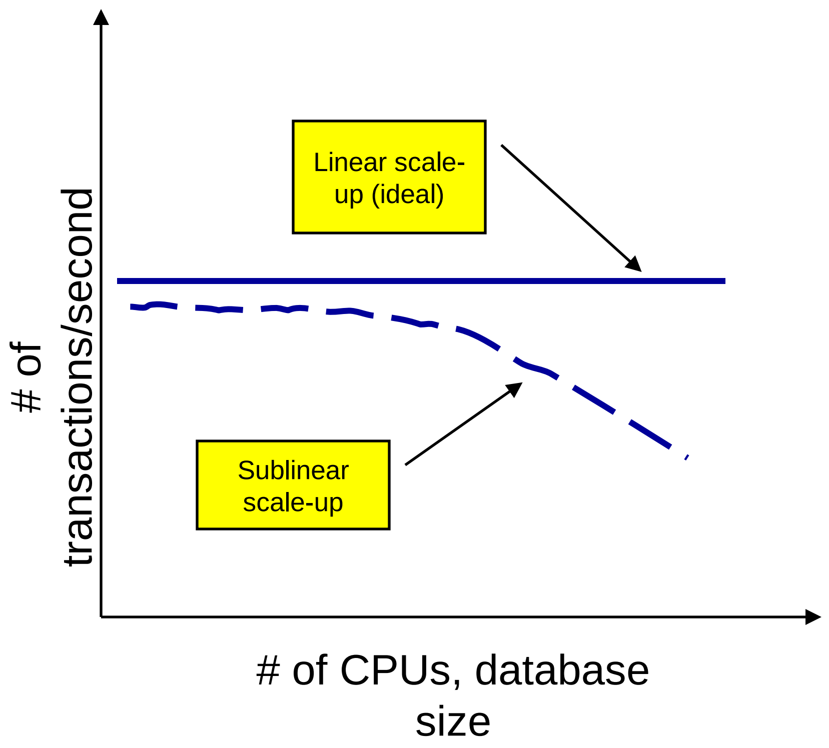
Multiprocessor Systems

- Linear speed-up occurs when the time required by an operation decreases in proportion to the increase in the number of CPUs and disks.
- Linear scale-up occurs when the performance level is sustained if the number of CPUs and disks are increased in proportion to the amount of data.
- As a result, ever-more-powerful parallel systems can be constructed by taking advantage of the rapidly improving performance for single-CPU systems and connecting as many CPUs as desired.

Speed-up vs. Scale-Up



<u>Speed-up</u>



<u>Scale-up</u>

Granularity

- A good metric for characterizing multiprocessors and placing then in context with other architectures is to consider the synchronization granularity, or frequency of synchronization, between processes in a system.
- Five categories of parallelism that differ in the degree of granularity can be defined:

1. Independent parallelism
2. Very coarse granularity
3. Coarse granularity
4. Medium granularity
5. Fine granularity

Independent Parallelism

- With independent parallelism there is no explicit synchronization among processes.
- Each process represents a separate, independent application or job.
- This type of parallelism is typical of a time-sharing system.
- The multiprocessor provides the same service as a multiprogrammed uniprocessor, however, because more than one processor is available, average response times to the user tend to be less.

Coarse and Very Coarse-Grained Parallelism

- With coarse and very coarse grained parallelism, there is synchronization among processes, but at a very gross level.
- This type of situation is easily handled as a set of concurrent processes running on a multi-programmed uni-processor and can be supported on a multiprocessor with little or no change to user software.
- In general, any collection of concurrent processes that need to communicate or synchronize can benefit from the use of a multiprocessor architecture.
- In the case of very infrequent interaction among the processes, a distributed system can provide good support. However, if the interaction is somewhat more frequent, then the overhead of communication across the network may negate some of the potential speedup. In that case, the multiprocessor organization provides the most effective support.

Medium-Grained Parallelism

- A single application can be effectively implemented as a collection of threads within a single process.
- In this case, the potential parallelism of an application must be explicitly specified by the programmer. Typically, there will need to be a rather high degree of coordination and interaction among the threads of an application, leading to a medium-grain level of synchronization.
- Whereas, independent, very-coarse, and coarse grain parallelism can be supported on either a multi-programmed uni-processor or a multiprocessor with little or nor impact on the scheduling function, we'll need to more carefully consider scheduling in the context of threads.
- Because the various threads of an application interact so frequently, scheduling decisions concerning one thread may affect the performance of the entire application.

Fine-Grained Parallelism

- Fine-grained parallelism represents a much more complex use of parallelism than is found in the use of threads.
- Although much work has been done on highly parallel applications, this is so far a specialized and fragmented area, with many different approaches.
- We will not be considering fine-grained parallelism any further.

Synchronization Granularity and Processes

Granularity	**Description**	**Synchronization Interval (Instructions)**
Fine	Parallelism inherent in a single instruction stream	< 20
Medium	Parallel processing or multitasking within a single application	20-200
Coarse	Multiprocessing of concurrent processes in a multiprogramming environment	200-2000
Very Coarse	Distributed processing across network nodes to form a single computing environment	2000- 10^6
Independent	Multiple unrelated processes	N/A

Design Issues For Multiprocessor Scheduling

- Scheduling on a multiprocessor involves three interrelated issues:

1. The assignment of processes to processors
2. The use of multiprogramming on individual processors
3. The actual dispatching of a process

- Looking at these three issues, it is important to keep in mind that the approach taken will depend, in general, on the degree of granularity of the applications and on the number of processors available.

1: Assignment of Processes to Processors

- If we assume that the architecture of the multiprocessor is uniform, in the sense that no processor has a particular physical advantage with respect to access to main memory or I/O devices, then the simplest scheduling protocol is to treat processors as a pooled resource and assign processes to processors on demand.
- The question then arises as to whether the assignment should be static or dynamic.

1: Assignment of Processes to Processors

- If a process is permanently assigned (static assignment) to a processor from activation until completion, then a dedicated short-term queue is maintained for each processor.
  - Allows for group or gang scheduling (details later).
  - Advantage: less overhead – processor assignment occurs only once.
  - Disadvantage: One processor could be idle (has an empty queue) while another processor has a backlog. To prevent this situation from arising, a common queue can be utilized. In this case, all processes go into one global queue and are scheduled to any available processor. Thus, over the life of a process, it may be executed on several different processors at different times.

1: Assignment of Processes to Processors

- In a tightly coupled shared-memory architecture, the context information for all processes will be available to all processors, and therefore the cost of scheduling a process will be independent of the identity of the processor on which it is scheduled.
- Yet another option to overcome this disadvantage is dynamic load balancing, in which threads are moved from a queue for one processor to the queue for another processor to balance the overall load on the various processors. Linux uses this approach.

1: Assignment of Processes to Processors

- Regardless of whether processes are dedicated to processors, some mechanism is needed to assign processes to processors.
- Two approaches have been used: master/slave and peer.

1. Master/slave architecture
   - Key kernel functions always run on a particular processor. The other processors can only execute user programs.
   - Master is responsible for scheduling jobs.
   - Slave sends service request to the master.
   - Advantages
     - Simple, requires little enhancement to a uniprocessor multiprogramming OS.
     - Conflict resolution is simple since one processor has control of all memory and I/O resources.
   - Disadvantages
     - Failure of the master brings down whole system
     - Master can become a performance bottleneck

1: Assignment of Processes to Processors

2. Peer architecture
   - The OS kernel can execute on any processor.
   - Each processor does self-scheduling from the pool of available processes.
   - Advantages:
     - All processors are equivalent.
     - No one processor should become a bottleneck in the system.
   - Disadvantage:
     - Complicates the operating system
     - The OS must make sure that two processors do not choose the same process and that the processes are not somehow lost from the queue.
     - Techniques must be employed to resolve and synchronize competing claims for resources.

2: The Use of Multiprogramming On Individual Processors

- The second of the design issues concerns the use of multiprogramming on the individual processors.
- When each process is statically assigned to a processor for the duration of its lifetime, a new question arises: Should that processor be multiprogrammed?
  - In a traditional multiprocessor, which is dealing with coarse-grain or independent synchronization granularity, it is clear that each individual processor should be able to switch among a number of processes to achieve high utilization and therefore better overall performance.
  - However, for medium-grained application running on a multiprocessor with many processors, the situation is less clear. When many processors are available, it is no longer paramount that every single processor be busy as much as possible. Rather, we are more concerned to provide the best performance, on average, for the applications. An application that consists of a number of threads may perform poorly unless all of its threads are available to run simultaneously.

3: Process Dispatching

- The final design issue related to multiprocessor scheduling is the actual selection of the process to run.
- Recall that on a multiprogrammed uniprocessor, the use of priorities or sophisticated scheduling algorithms based on past usage (SRT and HRRN) may improve performance over a FCFS protocol.
- When considering a multiprocessor environment, these complexities may be unnecessary or even counterproductive, and a simpler approach may be more effective with less overhead.
- In the case of thread scheduling, new issues come into play that may be more important than priorities or execution histories.
- We examine each of these issues separately.

Process Scheduling

- In most traditional multiprocessor systems, processes are not dedicated to processors. Rather there is a single queue for all processors, or if some sort of priority scheme is used, there are multiple queues based on priority all feeding into the common pool of processes.
- Much research has been conducted in this area and while many different conclusions have been determined, there is some consensus on the conclusions:
  - The specific scheduling protocol is much less important with two processors than with one.
  - The specific scheduling protocol becomes less and less important as the overall number of processors grows.
  - A simple FCFS protocol or FCFS within a static priority scheme typically suffices for a multiprocessor system.

Thread Scheduling

- With threads, the concept of execution is separated from the rest of the definition of the a process. An application can be implemented as a set of threads, which cooperate and execute concurrently in the same address space.
- On a uniprocessor, threads can be used as a program structuring aid to overlap I/O with processing. Because of the minimal penalty in doing a thread switch compared to a process switch, these benefits are realized with little cost.
- However, the full power of threads becomes evident in a multiprocessor system. In this environment, threads can be used to exploit true parallelism in an application.
- If the various threads of an application are simultaneously executed on separate processors, a dramatic gain in performance is possible.
- However, it has been shown that for applications that require significant interaction among threads (medium-grain parallelism), small differences in thread management and scheduling can have a significant performance impact.

Multiprocessor Thread Scheduling

- Among the many proposals for multiprocessor thread scheduling and processor assignment protocols, four general approaches stand out:

1. Load sharing
   - Processes are not assigned to a particular processor. A global queue of ready threads is maintained, and each processor, when idle, selects a thread from the queue. The term load sharing is used to distinguish this strategy from load balancing schemes in which work is allocated on a more permanent basis.
2. Gang scheduling
   - A set of related threads is scheduled to run on a set of processors at the same time, on a one-to-one basis.

Multiprocessor Thread Scheduling

3. Dedicated Processor Assignment
    - This is the opposite of the load-sharing approach and provides implicit scheduling defined by the assignment of threads to processors. Each program is allocated a number of processors equal to the number of threads in the program, for the duration of the program's execution. When the program terminates, the processors return to the general pool for possible allocation to another program.
4. Dynamic scheduling
    - The number of threads in a process can be altered during the course of execution.

Load Sharing

- Load sharing is perhaps the simplest approach and the one that carries over most directly from a uni-processor environment.
- Advantages:
  - The load is distributed evenly across the processors, assuring that no processor is idle while work is available to do.
  - No centralized scheduler is required; when a processor is available, the scheduling routine of the OS is executed on that processor to select the next thread for execution.
  - The global queue can be organized and accessed using any of the scheduling protocols we discussed for a uniprocessor environment, including priority-based protocols.

Load Sharing

- Disadvantages:
  - The central queue occupies a region of memory that must be accessed in a manner that forces mutual exclusion. Thus, it may become a bottleneck if many processors look for work at the same time. When there is only a small number of processors, this is unlikely to be noticeable. However, when the multiprocessor consists of dozens or even hundreds of processors, the potential for bottleneck is real.
  - Preempted threads are unlikely resume execution on the same processor. If each processor is equipped with a local cache, caching becomes less efficient.
  - If all threads are treated as a common pool of threads, it is unlikely that all of the threads of a program will gain access to processors at the same time. If a high degree of coordination is required between the threads of a program, the process switches involved may seriously compromise performance.

Load Sharing

- In spite of the disadvantages, load sharing is one of the most commonly used schemes in current multiprocessors.
- There are three common variants of load sharing that may be used:

1. FCFS (First Come First Served) – When a job arrives, each of its threads is placed consecutively at the end of the shared queue. When a processor becomes idle, it picks the next ready thread, which it executes until completion or it becomes blocked. Many simulations indicate this method is superior to the following two.
2. SNTF (Smallest Number of Threads First) – The shared ready queue is organized as a priority queue, with highest priority given to threads from jobs with the smallest number of unscheduled threads. Jobs of equal priority are ordered according to which job arrived first (FCFS). As with FCFS, a scheduled thread is run to completion or blocking.
3. PSNTF (Preemptive Smallest Number of Threads First) – Highest priority is given to jobs with the smallest number of unscheduled threads. An arriving job with a smaller number of threads than an executing job will preempt threads belonging to the scheduled job.

Gang Scheduling

- Gang scheduling (also referred to as group scheduling) is the simultaneous scheduling of all of the threads that make up a single process.
- Gang scheduling attempts to achieve the following benefits:
  - If closely related processes execute in parallel, synchronization blocking may be reduced, less process switching may be necessary, and performance will increase.
  - Scheduling overhead may be reduced because a single decision affects a number of processors and processes at one time.
- Gang scheduling is useful for medium or fine-grain parallel applications where performance severely degrades when any part of the application is not running while other parts are ready to run.
- It is also beneficial to any parallel application, even one that is not quite so performance sensitive.
- Threads often need to synchronize with each other

Gang Scheduling

- One obvious way in which gang scheduling improves the performance of a single application is that process switches are minimized.
  - Suppose one thread of a process is executing and reaches a point at which it must synchronize with another thread of the same process. If that other thread is not running, but is in a ready queue, the first thread is hung up until a process switch can be done on some other processor to bring in the needed thread.
- In an application with tight coordination among threads, such switches will dramatically reduce performance.
- The simultaneous scheduling of cooperating threads can also save time in resource allocation.
  - For example, multiple gang-scheduled threads can access a file without the additional overhead of locking during a seek or read/write operation.

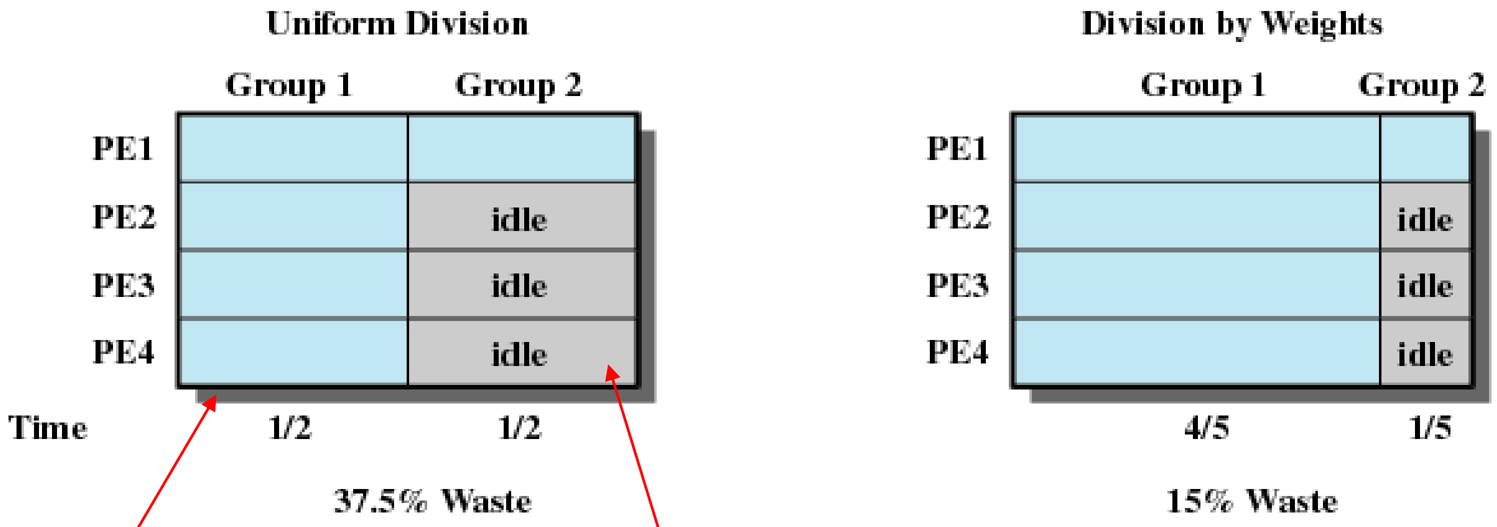
Gang Scheduling

- The use of gang scheduling creates the requirement for processor allocation.
- One possibility is the uniform allocation of time:
  - Suppose we have N processors and M applications, each of which has N or fewer threads. Then each application could be given 1/M of the available time on the N processors, using time slicing.
- The problem with this simple strategy is that it can be inefficient.
  - For example, suppose we have two applications, one with four threads and one with one thread. With four processors, the four threaded application keeps all four processors busy, but when the single threaded application is executed 3 of the 4 processors are idle. Uniform time allocation wastes 37.5% of the processing resource (3/8 of the total processing time is wasted). (See diagram on page 33.)

Gang Scheduling

- If there are several single-threaded applications, these could be ganged together to increase processor utilization.
- If that option is not available, an alternative to uniform time allocation is scheduling that is weighted by the number of threads.
- In the previous example, the four-threaded application would be allocated 4/5 of the time and the one-threaded application would be allocated 1/5 of the time. This would reduce processor waste to 15% (since 4/5 of the time all processors are busy and only 1/5 of the time are 3 processors idle). (See diagram on next page.)

Scheduling Groups



The 4-threaded application is assigned as Group1

The single-threaded application is assigned as Group2

Example of Scheduling Groups with Four and One Threads

Dedicated Processor Assignment

- An extreme form of gang scheduling, is to dedicate a group of processors to an application for the duration of the application. That is to say, when application is scheduled, its threads are assigned to a processor that remains dedicated to that thread until the application runs to completion.
- At first glance, this approach would appear to be extremely wasteful of processor time.
  - If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle.
  - There is no multiprogramming of processors. If a thread of an application is blocked waiting for I/O or for synchronization with another thread, then that thread's processor remains idle.

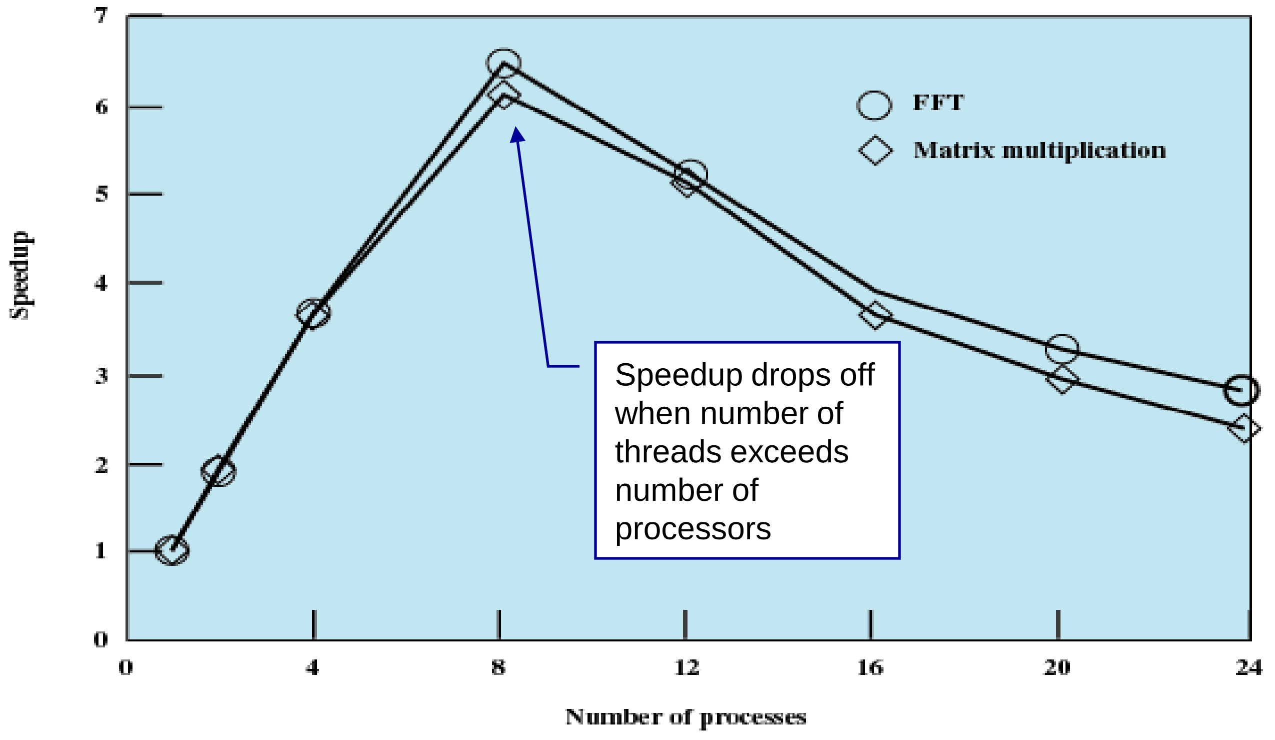
Dedicated Processor Assignment

- There are two observations regarding this extreme strategy that indicate better than expected performance:

1. In a highly parallel system, with tens or hundreds of processors, each of which represents a small fraction of the cost of the system, processor utilization is no longer an extremely important metric for effectiveness or performance.

2. Total avoidance of process switching during the lifetime of a program should result in a substantial speedup of that program.

Dedicated Processor Assignment



Speedup drops off when number of threads exceeds number of processors

Test results for multiprocessor system with 16 processors

Application Speedup as a Function of Number of Processes

Dynamic Scheduling

- For some applications, it is possible to provide language and system tools that permit the number of threads in the process to be altered dynamically.
- This allows the OS to adjust the load to improve utilization.
- In this technique the OS and the application cooperate in making scheduling decisions.
- The OS is responsible for partitioning the processors among the various jobs.
- Each job uses the processors currently in its partition to execute some subset of its runnable tasks by mapping these tasks to threads.
- An appropriate decision about which subset to run as well as which thread to suspend when a process is preempted, is left to the individual applications (probably through a set of run-time library routines).

Dynamic Scheduling

- This approach may not be suitable for all applications. However, some applications could default to a single thread while others could be programmed to take advantage of this particular feature of the OS.
- Analysis has shown that for applications that can take advantage of dynamic scheduling, the approach is superior to gang scheduling or dedicated processor assignment.
- However, the overhead of this approach may negate this apparent performance advantage.
- In this approach, the scheduling responsibility of the OS is primarily limited to processor allocations and proceeds according to the following protocol:

Dynamic Scheduling

- When a job requests one or more processors (either when the job arrives for the first time or because its requirements change),

  1. If there are idle processors, use them to satisfy the request.
  2. Otherwise, if the job making the request is a new arrival, allocate it a single processor by taking one away from any job currently allocated more than one processor.
  3. If any portion of the request cannot be satisfied, it remains outstanding until either a processor becomes available for it or the job rescinds the request (e.g., there is no longer a need for the extra processors).

  Upon release of one or more processors (including job departure),

  4. Scan the current queue of unsatisfied requests for processors. Assign a single processor to each job in the list that currently has no processors (i.e., to all waiting new arrivals). Then scan the list again, allocating the remaining processors on a FCFS basis.