OS

Chapter – 2 "Process Management"

Topics Covered

1) What is thread?

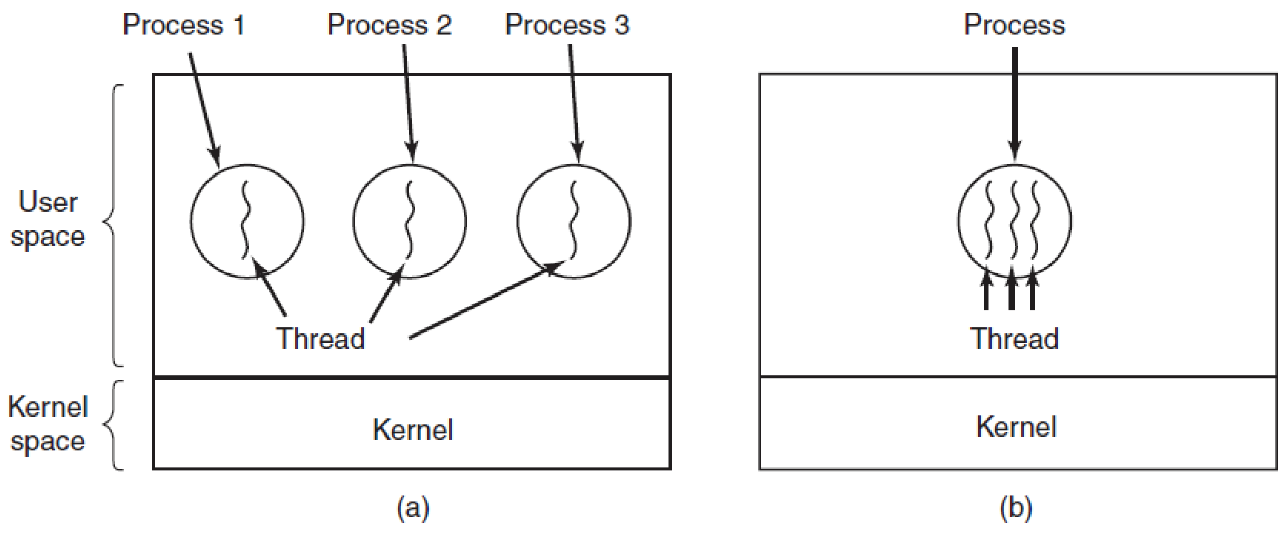
- In traditional operating systems, each process has an address space and a single thread of execution.
- In many situations, it is desirable to have multiple threads of control in the same address space running in parallel, as though they were separate processes (except for the shared address space).
- "Thread is the smallest unit of processing that can be scheduled by an operating system".
- Because threads have some of the properties of processes, they are sometimes called "lightweight processes".

2) Thread Structure

- The thread has
  - a program counter that keeps track of which instruction to execute next.
  - registers, which holds its current working variables.
  - stack, which contains the execution history, with one frame for each procedure called but not yet returned from.
- "A thread must execute in some process". But the thread and its process

are different concepts and can be treated separately.

- What threads add to the process model is to allow multiple executions to take place in the same process environment (same address space), to a large degree independent of one another.
- Having multiple threads running in parallel in one process is similar to having multiple processes running in parallel in one computer.



(a) Three processes each with thread. *(b) One process with three one threads.*

- In Fig. (a) we see three traditional processes. Each process has its own address space and a single thread of control. Each of them operates in a different address space.
- In Fig. (b) we see a single process with three threads of control. all three of them share the same address space.
- Like a traditional process (i.e., a process with only one thread), a thread can be in any one of several states: new, ready, running, blocked, or terminated.

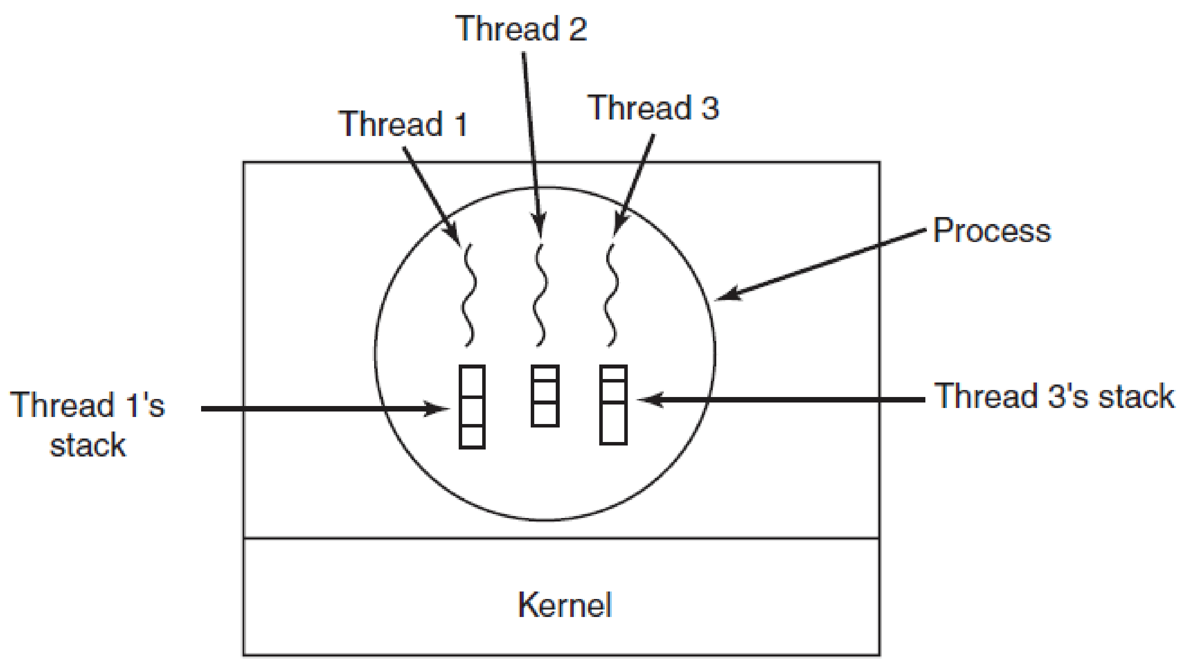


Fig : Each thread has its own stack

3) Types of threads

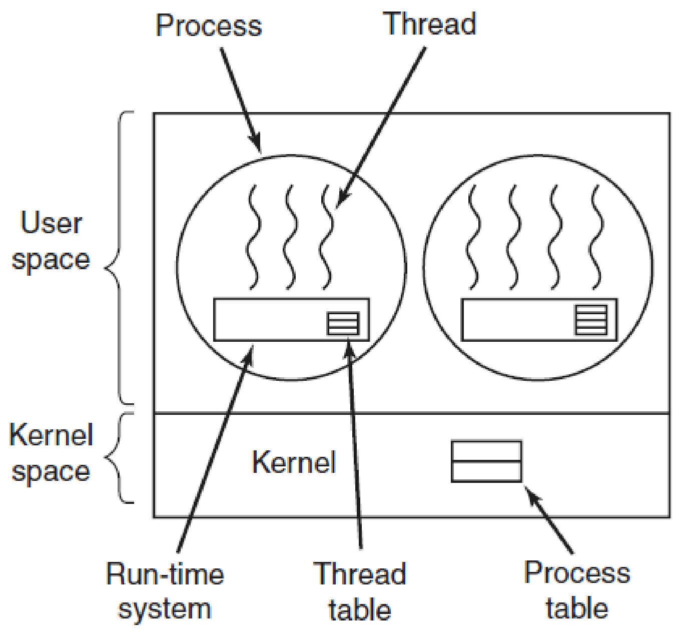
There are 2 Types of threads

a) User Level Threads

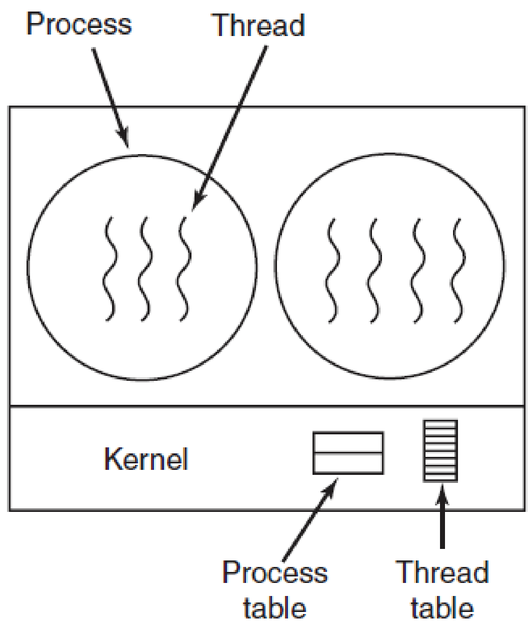
b) Kernel Level Threads

a) User Level Threads

- User level threads are implemented in user level libraries, rather than via systems calls.
- So thread switching does not need to call operating system and to cause interrupt to the kernel.
- The kernel knows nothing about user level threads and manages them as if they were single threaded processes.
- When threads are managed in user space, each process needs its own private thread table to keep track of the threads in that process.
- This table keeps track only of the thread properties, such as each thread's program counter, stack pointer, registers, state, and so forth.
- The thread table is managed by the run-time system.



a) User – level thread package



b) Kernel - level thread package

Advantages of User level Threads

1. Simple Representation: Each thread is represented simply by a PC, registers, stack and a small control block, all stored in the user process address space.
2. Simple Management: This simply means that creating a thread, switching between threads and synchronization between threads can all be done without interference of the kernel.
3. Fast and Efficient: Thread switching is not much more expensive than a procedure call.
4. It can be implemented on an Operating System that does not support threads.
5. A user level thread does not require modification to operating systems.
6. They allow each process to have its own customized scheduling algorithm.

Disadvantages of User Level Thread

1. There is a lack of coordination between threads and operating system kernel. Therefore, process as a whole gets "one time slice" irrespective of whether process has one thread or 1000 threads within. It is up to each

thread to give up control to other threads.

2. Another problem with user-level thread packages is that if a thread starts running, no other thread in that process will ever run unless the first thread voluntarily gives up the CPU.
3. A user level thread requires non-blocking systems call (i.e., a multithreaded kernel). Otherwise, entire process will be blocked in the kernel, even if a single thread is blocked but other runnable threads are present.

b) Kernel Level Threads

- In this method, the kernel knows about threads and manages the threads.
- Instead of thread table in each process, the kernel has a thread table that keeps track of all threads in the system. In addition, the kernel also maintains the traditional process table to keep track of processes.
- Operating Systems kernel provides system call to create and manage threads.

Advantages of Kernel Level Threads

1. Because kernel has full knowledge of all threads, scheduler may decide to give more time to a process having large number of threads than process having small number of threads.
2. Blocking of one thread in a process will not affect the other threads in the same process as Kernel knows about multiple threads present so it will schedule other runnable thread.
3. So, Kernel threads do not require any new, non-blocking system calls.

Disadvantages of Kernel Level Threads

1. The kernel level threads are slow and inefficient. As thread are managed by system calls, at considerably greater cost.
2. Since kernel must manage and schedule threads as well as processes. It requires a full Thread Control Block (TCB) for each thread to maintain

information about threads.
3. As a result there is significant overhead and increased in kernel complexity.

4) Difference between Process and threads

User Level Threads	**Kernel Level Threads**
1) The user level threads are faster and easier to create and manage.	1) The kernel-level threads are slower and complex to create and manage.
2) Supported above the kernel and implemented by thread library at user level.	2) Directly supported from the OS.
3) Can run on any OS.	3) Specific to the OS.
4) A library provides support for thread creation , scheduling and management with no support from kernel.	4) Thread creation , scheduling and management is done in kernel space with the help of OS directly.
5) Kernel is unaware of User level threads.	5) Kernel has full knowledge of these threads.
6) Process as a whole gets "one time slice" irrespective of whether process has one thread or 1000 threads within.	6) Scheduler may decide to give <u>more time</u> to a process having large number of threads than process having small number of threads.

5) Difference between Kernel level thread and user level thread

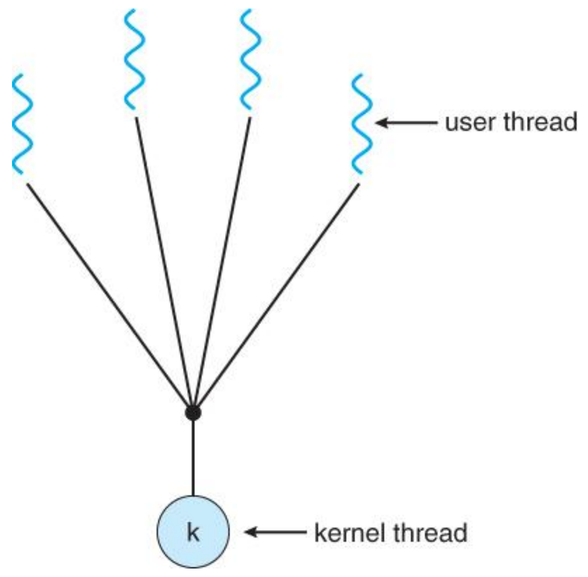
Process	Thread
1) Process is heavy weight and uses more resources.	1) Thread is light weight and uses less resources than a process.
2) Process switching needs interaction with operating system	2) Thread switching does not need to interact with operating system.
3) All the processes have separate address space.	3) All the threads of a process share common address space.
4) Process switching is more expensive .	4) Thread switching is less expensive.
5) Processes are complex to create because it requires separate address space.	5) Threads are easier to create because they don't require separate address space.
6) If one process terminates, its resources are deallocated and all its threads are terminated.	6) If a thread in a process terminates, another thread of the same process can run.
7) Processes can live on its own.	7) Threads can't live on its on. It must lie within a process.

8) Multithreading Models

- Support for threads may be provided either at the user level, for user threads, or by the kernel, for kernel threads.
- User threads are supported above the kernel and are managed without kernel support, whereas kernel threads are supported and managed directly by the operating system.
- Ultimately, there must exist a relationship between user threads and kernel threads. There are three common ways of establishing this relationship.
    1) Many-to-One Model
    2) One-to-One Model
    3) Many-to-Many Model

1) Many-to-One Model:

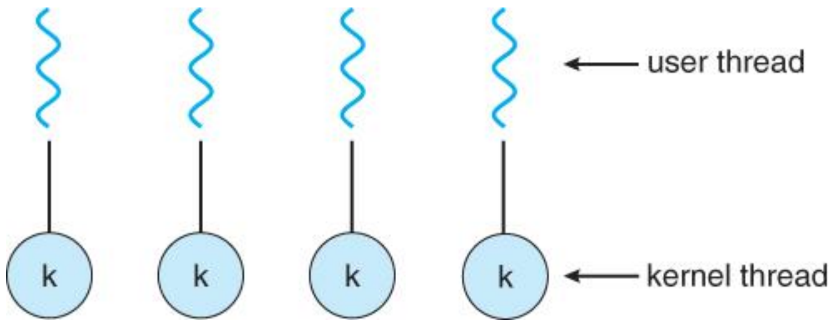
- The many-to-one model maps many user-level threads to one kernel thread.



- Also, because only one thread can access the kernel at a time, multiple threads are unable to run in parallel on multiprocessors.
- Thread management is done by the thread library in user space, so it is efficient; but the entire process will block if a thread makes a blocking system call.

2) One-to-One Model

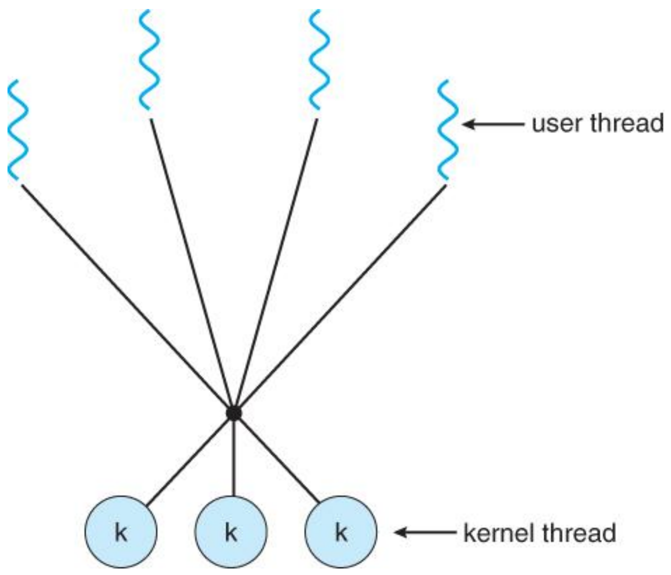
- The one-to-one model maps each user thread to a kernel thread.



- It provides more concurrency than the many-to-one model by allowing another thread to run when a thread makes a blocking system call;
- It also allows multiple threads to run in parallel on multiprocessors. The only drawback to this model is that creating a user thread requires creating the corresponding kernel thread.
- Because the overhead of creating kernel threads can burden the performance of an application, most implementations of this model restrict the number of threads supported by the system.
- Example : Linux, along with the family of Windows operating systems—including Windows 95, 98, NT, 2000, and XP implement the one-to-one model.

3) Many-to-Many Model

- The many-to-many model multiplexes many user-level threads to a smaller or equal number of kernel threads.
- The number of kernel threads may be specific to either a particular application or a particular machine.
- Developers can create as many user threads as necessary, and the corresponding kernel threads can run in parallel on a multiprocessor. Also, when a thread performs a blocking system call, the kernel can schedule another thread for execution.



6) Types of scheduler

1) Long Term Scheduler

- It is a first level scheduler.
- It is also called as a 'Job Scheduler'.
- It is found in OS where there is a support for 'batch', like batch OS.
- It works with batch queue and selects next batch of jobs to be executed.
- Jobs contain all the necessary data and commands for its execution. Jobs also contains resources such as memory and I/O devices.
- Primary objective of long term scheduler is to provide balanced mix of jobs such as Processor bound and I/O bound.
- When processor utilization is low, the scheduler may admit more jobs to increase the no. of processes in ready queue.
- When processor utilization is high, the scheduler may try to reduce the rate of processes in ready queue.
- It is much slower than other two types of schedulers.

2) Medium Term Scheduler

- It is a second level scheduler.
- It is found in OS where there is support for swapping.
- Running process may become suspended if it makes an I/O request.
- Suspended processes cannot make any progress towards completion.
- In this condition, to remove the process from memory and make space for other process, the suspended process is moved to the secondary storage.
- This process is called swapping, and the process is said to be 'swapped out' .
- However, suspending condition is removed, the medium term scheduler attempts to allocate such processes in main memory, which is known as 'swap in'.
- Thus, its main functionality is to swap in and swap out processes between main memory and disk.
- It's main objective is to make efficient use of memory.

3) Short Term Scheduler

- It is third level of scheduler.
- It can be found in all modern OS.
- It is also known as 'dispatcher' or 'CPU scheduler'.
- This scheduling makes scheduling decisions much more frequently than long term and medium term scheduler.
- Main objective is increasing system performance in accordance with the chosen set of criteria.
- CPU scheduler selects process among the processes that are ready to execute and allocates CPU to one of them.
- Short term scheduler is faster than long term and medium term schedulers.

7) Multiprocessor scheduling

- On a uniprocessor, scheduling is one dimensional. The only question that must be answered is "Which process to run next?"
- On a multiprocessor, scheduling is two dimensional. The scheduler has to decide "which process to run next?" and "Which CPU to run it on?"
- This extra dimension greatly complicates scheduling on multiprocessors.

1) Time sharing

- Let's first consider the case of scheduling independent processes; later we will consider how to schedule related processes.
- A single data structure can be used for scheduling a multiprocessor.

Example

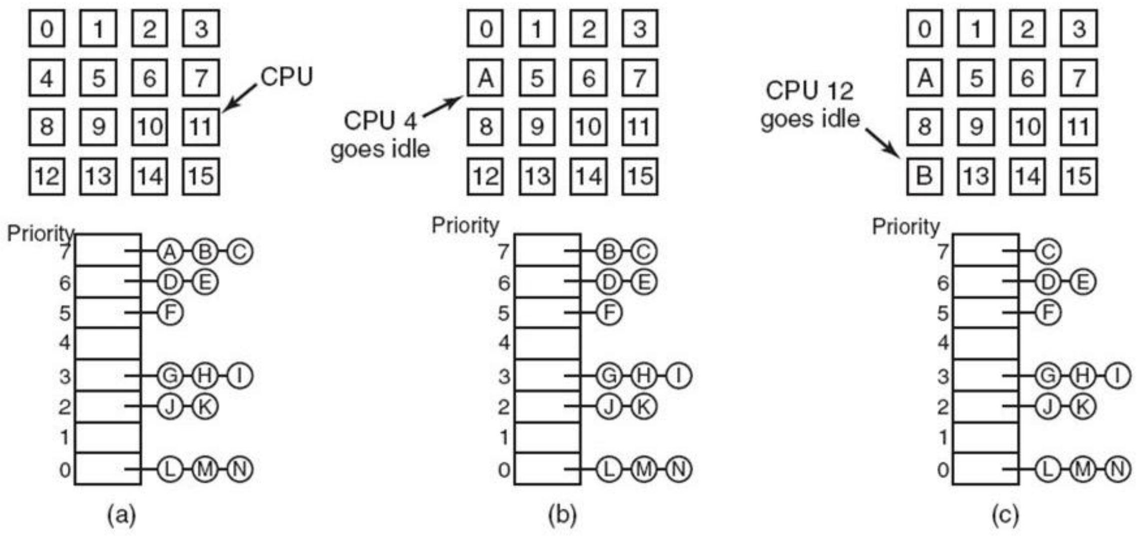


Fig. 1 using a single data structure for scheduling a multiprocessor

- Here the 16 CPUs are there in multiprocessor system. And all the processors are currently busy, and 14 processes A to N are waiting.
- Suppose that CPU 4 becomes free, it will choose the highest priority process A, as shown in Fig. 1.
- Next CPU 12 goes idle and chooses process B.
- As long as the processors are completely unrelated, doing scheduling this

way is a reasonable choice.

- This method provides automatic load balancing. Because it can never happen that one CPU is idle while others are overloaded.
- One issue that plays a role in scheduling is the fact that while all CPUs are equal, some CPUs are more equal.
- In particular, when process A has run for a long time on CPU K, CPU K’s cache will be full of A's blocks.
- If A gets to run again soon, it may perform better if it is run on CPU K, because K's cache may still contain some of A's blocks.
- Having cache blocks preloaded will increase the cache hit rate and thus the thread’s speed.
- In addition, the TLB may also contain the right pages, reducing TLB faults.
- Some multiprocessors take this effect into account and use what is called “affinity scheduling”.
- Affinity scheduling :
    - “The basic idea here is to make a serious effort to have a process run on the same CPU it run on last time.”
    - One way to create this affinity is to use a “two level scheduling algorithm”.
- Two level scheduling algorithm :
    - When a process is created, it is assigned to a CPU. This assignment of processes to CPU is the <u>top level</u> of the algorithm.
    - As a result, each CPU acquires its own collection of processes.
    - The <u>bottom level</u> of the algorithm is actual scheduling of the processes. It is done by each CPU separately using priorities or some other means.
    - By trying to keep a process on the same CPU, cache affinity is maximized.
    - However if a CPU has no processes to run, it takes one from another CPU rather than go idle.

- Benefits of Two level scheduling algorithm :
    1) It distributes the load evenly over the available CPUs.
    2) Second advantage is taken of cache affinity where possible.
    3) Third, by giving each CPU its own ready list, contention for the ready lists is minimized because attempts to use another CPU's ready list are relatively infrequent.

2) SPACE SHARING

- The other general approach to multi-processor scheduling can be used when processes are related to one another in same way.
- It often occurs that a single process creates multiple threads that work together. We will refer to the schedulable entities as threads (but the material holds for the process as well).
- "Scheduling multiple threads at the same time across multiple CPUs is called space sharing."
- The simplest space sharing algorithm works like this. Assume that the entire group of related threads is created at once.
- The scheduler checks to see if there are as many free CPUs as there are threads. If there are, each thread is given to its own dedicated (i.e. multi programmed) CPU and they all start.
- If there are not enough CPUs, none of them are started until enough CPUs are available.
- Each thread holds onto its CPU until it terminates, (at which time the CPU is put back into the pool of available CPUs.)
- If a thread blocks on I/O, it continues to hold the CPU, which is simply idle until the thread wakes up.
- At any instance of time, the set of CPUs is statically partitioned into some no. Of partitions, each one running the threads of one process.

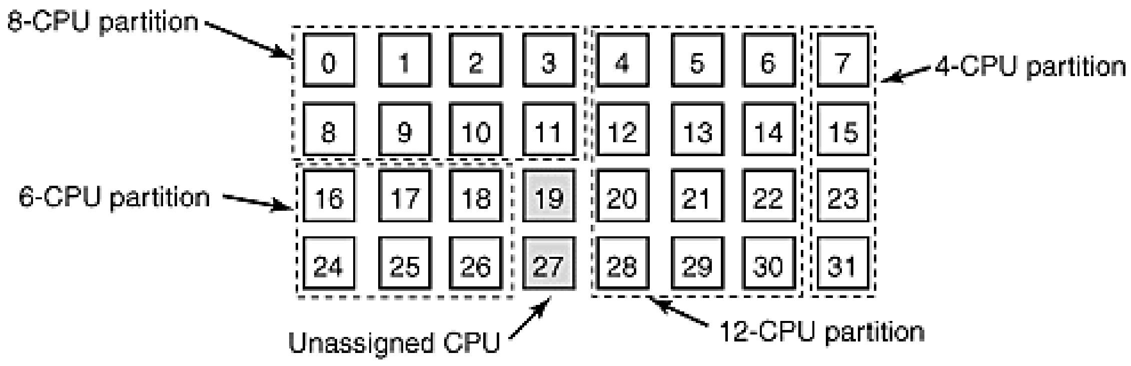


Fig. 2 Using a single data structure for scheduling a multiprocessor

- In Fig. 2, A set of 32 CPUs split into 4 partitions, with two CPUs, available.
- We have partitions of sizes 4, 6, 8, 12 CPUs with 2 CPUs unassigned.
- As the time goes the number and size of partitions will change as processes come and go.
- Periodically, scheduling decisions have to be made. In uniprocessor systems, Shortest Job First is a well known algorithm for Batch Scheduling.
- In this Simple partitioning model, a process just ask for some number of CPUs, and either gets them all or has to wait until they are available.
- A different approach is for processes to actively manage the degree of parallelism.
- One way is to have a central server that keeps track of which processes are running and want to run and what their minimum and maximum CPU requirements are.
- Periodically, each CPU pulls the central server to ask how many CPUs it may use.

Example

- A web server can have 1, 2, 5, 10, 20 or any other number of threads running in parallel.
- If it currently has 10 threads and there is suddenly more demand for CPUs and it is told to drop 5, when the next 5 threads finish their current works, they are told to exit instead of being given new work.

- This scheme allows the partition sizes to vary dynamically to match the current workload.

GANG SCHEDULING

- A clear advantage of space sharing is the elimination of multiprogramming which eliminates the context switching overhead.
- However, an equally clear disadvantage is the time wasted when a CPU blocks and has nothing to do until it becomes ready again.

Problem:

- Let's see the kind of problem that can arise when the threads of a process are independently scheduled.
- Consider a system with threads A0 and A1 belonging to process A and threads B0 and B1 belonging to process B.
- Threads A0 and B0 are time shared on CPU 0.
- Threads A1 and B1 are time shared on CPU 1.
- Now threads A0 and A1 needs to communicate often.
- The communication pattern is that A0 sends A1 a message, and A1 sends back a reply to A0 followed by another such sequence.
- Suppose A0 and B1 starts first.
- In time slice 0, A0 sends A1 a request but A1 doesn't get it until it runs in time slice 1 starting at 100ms. It sends a reply immediately but A0 doesn't get it until it runs again at 200ms.
- The next result is one request-replay sequence every 200ms not very good.

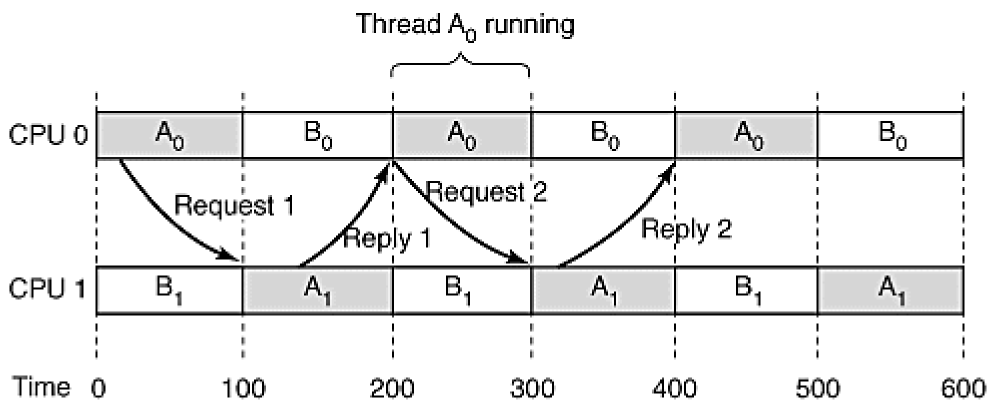


Fig. 3 Using a single data structure for scheduling a multiprocessor

Solution :

The solution to this problem is gang scheduling. Which has three parts:

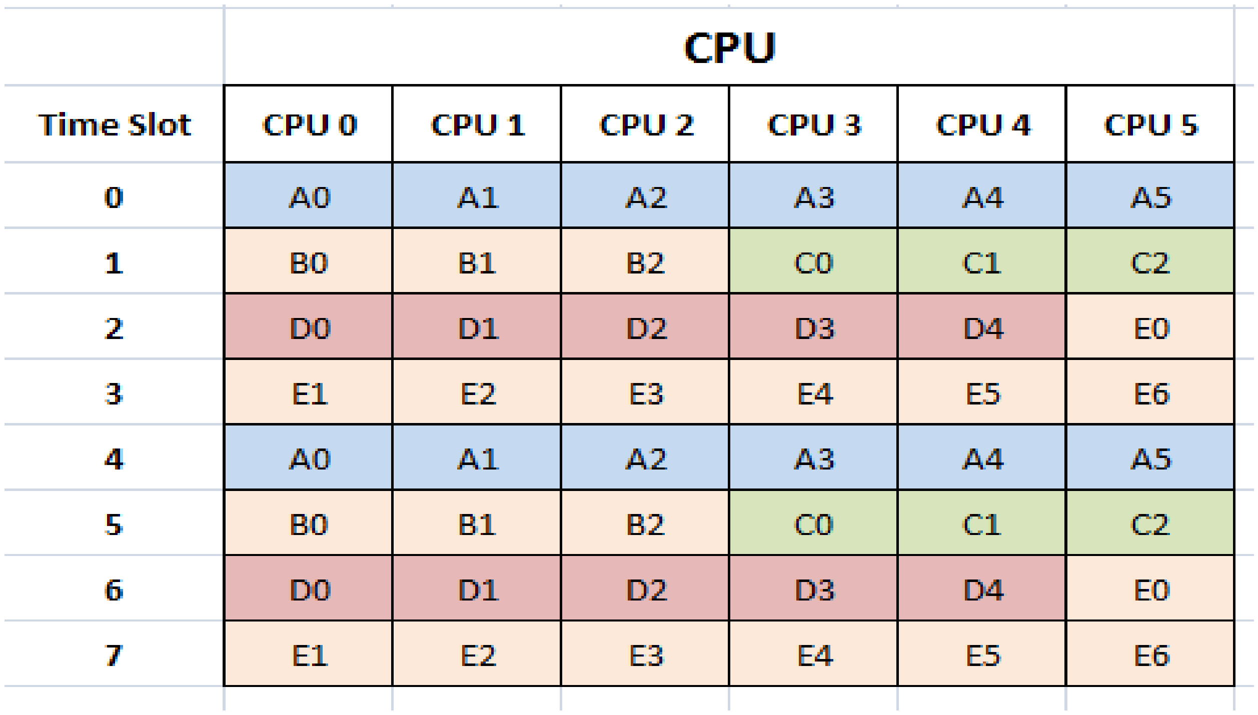
1) Groups of selected threads are scheduled as a unit, a gang.

2) All members of a gang run simultaneously on different timeshared CPUs.

3) All gang members start and end their time slice together.

- The trick that makes gang scheduling work is that all the CPUs are scheduled synchronously.
- At the start of each new quantum, all the CPUs are rescheduled, with a new thread being started on each one.
- If a thread blocks its CPU stays idle until the end of the quantum.

Example:



	CPU					
Time Slot	CPU 0	CPU 1	CPU 2	CPU 3	CPU 4	CPU 5
0	A0	A1	A2	A3	A4	A5
1	B0	B1	B2	C0	C1	C2
2	D0	D1	D2	D3	D4	E0
3	E1	E2	E3	E4	E5	E6
4	A0	A1	A2	A3	A4	A5
5	B0	B1	B2	C0	C1	C2
6	D0	D1	D2	D3	D4	E0
7	E1	E2	E3	E4	E5	E6

Fig. 4 Using a single data structure for scheduling a multiprocessor

- 6 CPUs used by 5 processes A through E. 24 total Ready Threads.
- During time slot 0, threads A0 , A1, A2, A3, A4, and A5 are scheduled to run.
- During time slot 1, threads B0, B1, B2, C0, C1 and C2 are scheduled to run.

- During time slot 2, *D*'s five threads and *E*0 get to run. The remaining six threads belonging to thread *E* run in time slot 3.
- Then the cycle repeats, with slot 4 being the same as slot 0 and so on.
- The idea of gang scheduling is to have "All the threads of a process run together, at the same time, on different CPUs, so that if one of them sends a request to another one, it will get the message almost immediately and be able to reply almost immediately".
- In Fig. 4 since all the *A* threads are running together, during one quantum, they may send and receive a very large number of messages in one quantum, thus eliminating the problem of Fig. 3.