Multiprocessor, Multicore, and Realtime Scheduling

Arnav Gupta

April 7, 2024

Contents

1 Multiprocessor and Multicore Scheduling

Multiprocessor systems can be classified as:

- **loosely coupled or distributed multiprocessor or cluster**: has collection of relatively autonomous systems, each processor having its own main memory and IO channels
- **functionaly specialized processors**: controlled by the master processor and provide services to it
- **tightly coupled multiprocessor**: has set of processors that share a common main memory and are under the integrated control of an OS

1.1 Granularity

Multiprocessors can be categorized and placed in context with other architectures by considering **synchronization granularity** or frequency between processes in a system.

Grain categories are:

- **fine**: parallelism inherent in a single instruction stream, interval <20 instructions
- **medium**: parallel processing or multitasking within a single application, interval 20-200 instructions
- **coarse**: multiprocessing of concurrent ptocesses in a multiprogramming environment, interval 200-2000 instructions
- **very coarse**: distributed processing across network nodes to form a single computing environment, interval 2000-1 million instructions
- **independent**: multiple unrelated processes, interval not applicable

1.1.1 Independent Parallelism

No explicit synchronization between processes, each represents a separate job. Example is time-sharing system. Works similar to multiprogrammed uniprocessor with shorter average response time.

1.1.2 Coarse and Very Coarse-Grained Parallelism

Only high-level synchronization among processes so can work as concurrent processes running on a multiprogrammed uniprocessor, so not much user code change.

Any collection of concurrent processes that need to communicate or synchronize can benefit from multiprocessor use. If infrequent process interaction, a distributed system could provide support, if frequent, then overhead of network communication negates some speedup.

1.1.3 Medium-Grained Parallelism

If an application is a collection of threads in a process, the programmer must specify potential parallelism. Typically, much coordination and interaction among the threads of an application.

Since application threads interact frequently, scheduling decisions concerning 1 thread may affect application performance.

1.1.4 Fine-Grained Parallelism

More complex use of parallelism found in the use of threads. Specialized and fragmented area.

1.2 Design Issues

3 interrelated issues for multiprocessor scheduling:

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

1.2.1 Assignment of Processes to Processors

Assume multiprocessor architecture is uniform and treat processors as a pooled resource and assign processes to processors on demand.

With static assignment, a processor is permanently assinged to one processor from activation to completion, so a dedicated short-term queue is maintained for each processor. Advantage is less overhead in scheduling since processor assignment is known, allows for gang scheduling. Disadvantage is uneven load (one processor empty, one has backlog). To fix this, a global queue can be used. Context info for all processes will be available to all processors and so the cost of scheduling will be independent of the identity of the processor on which it is scheduled. Another option is dynamic load balancing, where threads are moved from a queue for one processor to a queue for another processor.

Approaches to assign processes to processors are master/slave and peer.

With master/slave, kernel functions like scheduling are only on a particular processor (master) and slaves run user programs and call master for service calls (IO). Conflict resolution simple since 1 processor has control of memory and IO resources. Disadvantages are the danger of master failure and bottlenecks from the master.

With peer architecture, any processor can run kernel and each process self-schedules. Must ensure processes aren't lost and 2 processors do not choose the same process.

1.2.2 Use of Multiprogramming on Individual Processes

When many processors are available, no need for every processor to be as busy as possible but rather provide best average performance.

An application with many threads may run poorly unless all can run simultaneously.

1.2.3 Process Dispatching

For multiprocessors, priority or scheduling algorithms may have unnecessary overhead. For thread scheduling, new issues are in play that may be more important than priorities or execution histories.

1.3 Process Scheduling

Most traditional multiprocessor systems have common pool of processors.

Result of speedup of scheduling algorithms depends on variability in service times, measured with coefficient of variation C_s. Larger C_s means more variation. With 2 processors, longer processes do not hold up FCFS too long.

Specific scheduling discipline is less important with 2+ processors than 1. Simple FCFS (possible with priorities) may work for a multiprocessor system.

1.4 Thread Scheduling

On a uniprocessor, threads help with program structure and overlapping IO with processing.

In a multiprocessing environment, threads allow parallelism leading to dramatic gains in performance. With medium-grained parallelism, small differences in thread management and scheduling can have significant performance impact.

General approaches for multiprocessor thread scheduling and processor assignment:

1. **load sharing**: global queue of ready threads and each processor selects a thread from the queue when idle
2. **gang scheduling**: a set of related threads is scheduled to run on a set of processors at the same time on a 1:1 basis
3. **dedicated processor assignment**: threads assigned to processors such that each thread in a program gets one processor and when program terminates, all processors returned to general pool for allocation to another program
4. **dynamic scheduling**: number of threads in a process can be altered during execution

1.4.1 Load Sharing

Advantages:

- load distributed evenly across processors, no unnecessarily idle processor
- no centralized scheduler needed, scheduling run on available processors
- global queue organized and accessed using any scheduling policy

Versions of load sharing:

1. **FCFS**: when a job arrives, each thread placed consecutively at the end of the shared queue, idle processors pick the next ready thread
2. **smallest number of threads first**: shared ready queue is organized as a priority queue with highest priority given to threads from jobs with least unscheduled threads, if equal then arrival time used

3. **preemptive smallest number of threads first**: highest priority given to jobs with least unscheduled threads, and a new job with less threads than the current can preempt threads belonging to the scheduled job

FCFS is superior since gang scheduling better than load sharing.

Disadvantages of load sharing are:

- central queue has a region of memory that must enforce mutual exclusion, which could lead to a bottleneck and could be a problem if not enough processors
- preempted threads unlikely to resume execution on the same process, so caching less efficient
- unlikely all threads gain access to processors at the same time so much coordination required

1.4.2 Gang Scheduling

The concept of scheduling a set of processes simultaneously on a set of processors.

Advantages:

- if processes in the group are related or coordinated, synchronization blocking could be reduced, less process switching, and better performance
- single scheduling decision affects multiple processors and processes, so less scheduling overhead

Coscheduling: scheduling a related set of tasks (task force) that are small and close to the idea of a thread

Useful for medium and fine-grained applications where performance degrades when any part of the application is not running while others are ready to run.

Gang scheduling reduces process switches since all threads are in system and saves time in resource allocation since process has IO access.

If there are several single-thread applications, they could all fit together to increase processor utilization. Otherwise, scheduling could be weighted by the number of threads.

1.4.3 Dedicated Processor Assignment

When an application is sceheduled, each thread is assigned a processor that is dedicated to that thread until application completion.

Wastes time in the case of blocking for IO or synchronization. Defence of this strategy:

1. in a highly parallel system, processor utilization is no longer an important metric for effectiveness or performance
2. avoidance of process switching during program lifetime results in speedup

Performance degrades as both total number of processes excepts number of processors. More threads means more thread preemption and rescheduling, which leads to inefficiency from many sources. Limit the number of active threads to the number of processors.

Activity working set is the minimum number of threads that must be scheduled simultaneously on processors for the application to make acceptable progress. Not scheduling activity working set could lead to processor thrashing.

Processor thrashing: when scheduling threads whose services are required deschedules other threads whose services are soon needed

Processor fragmentation: situation in which some processors are left over when others are allocated and leftovers have insufficient number or lack organization to support waiting applications

Gang scheduling and dedicated processor allocation avoid these

1.4.4 Dynamic Scheduling

Number of threads in process can be altered dynamically, so OS can adjust load to improve utilization.

OS partitions processors among jobs and each job executes subset of tasks by mapping them to threads (subset decided by application).

Here, scheduling responsibility of OS is limited to processor allocation. When a job requests 1+ processors (when job arrives for the 1st time or requirements change):

1. use idle processors first to satisfy request

2. if the requesting job is new, allocate a single processor by taking it from any job currently allocated 1+ processors
3. if any portion of the request cannot be satisfied, it remains outstanding until a process becomes available or the job rescinds the request
4. when 1+ processors are released, scan the current queue for unsatisfied requests and assign a single processor to each job in the list with no processors, then scan again and allocate based on FCFS

Dynamic allocation superior to gang scheduling and dedicated processor assignment, but overhead may negate performance advantage.

1.5 Multicore Thread Scheduling

Some OSs treat multicore the same as multiprocessor systems (use load balancing).

More cores per chip means minimize access to off-chip memory (with cache and locality) rather than maximize processor utilization. This could be complex for caches shared by only some cores. For this case, scheduling should assign threads that share memory resources to those that share cache and vice versa for load balancing.

Two aspects of cache sharing:

- **cooperative resource sharing**: multiple threads access the same set of main memory locations
  - data brought into a cache by one thread must be accessed by a cooperating thread, so cooperating threads should be on adjacent cores (share cache)
- **resource contention**: when threads compete for cache memory locations
  - if more cache is allocated to one thread, other thread(s) suffer performance degradation
  - with contention-aware scheduling, allocate threads to cores to maximize effectiveness of shared cache memory

2 Real-Time Scheduling

2.1 Background

Real-time computing: computing where system correctness depends on logical results and time when results are produced

Real-time tasks attempt to control or react to events in the outside world, with which it must keep up.

Hard real-time task must meet its deadline or cause unacceptable damage or fatal error to the system

Soft real-time task has an associated deadline that is desirable but not mandatory, can still be scheduled and completed after its deadline

Aperiodic task has a deadline by which it must finish or start or may have a constraint on start and finish time

Periodic task has a requirement every T units apart

2.2 Characteristics of Real-Time Operating Systems

Have unique requirements in the areas of determinism, responsiveness, user control, reliability, and fail-soft operation.

An OS is **deterministic** based on the extent it performs operations at fixed, predetermined times or within predetermined time intervals. Impossible when processes compete for resources and in an RTOS, process requests for service are dictated by external events and timings. Determinism depends on interrupt response speed and request capacity per time. One measure of determinism is delay from high priority interrupt to service.

Responsiveness is concerned with how long after acknowledgement it takes an OS to service the interrupt. Aspects include:

1. amount of time required to handle the interrupt and begin execution of the interrupt service routine (ISR), if this requires a process switch then longer delay

2. amount of time to perform ISR, depends on hardware

3. effect of interrupt nesting, service delayed if ISR can be interrupted

User control is essential in real-time systems to allow users fine-grained control over task priority. User should be able to distinguish between hard and soft tasks and specify relative priorities for each. Real-time systems allow the user to specify memory use as well (paging, swapping, resident set, etc).

Reliability is important since real-time systems respond to and control events in real time. Performance degradation may have catastrophic consequences.

Fail-soft operation: the ability of a system to fail in such a way as to preserve capability and data.

Real-time systems will not simply fail but attempt to correct or minimize effect of failure and continue to run. User is notified of corrective action and reduced level of service. If shutdown, data consistency maintained.

A real-time system is stable if the system at least meets the deadlines of its most critical, highest-priority tasks.

Important features for real-time OSs:

- a stricter use of priorities with preemptive scheduling designed to meet real-time requirements
- interrupt latency (time between generation and servicing) bounded and relatively short
- precise and predictable timing characteristics

Design short-term scheduler so that all hard real-time tasks complete (or start) by their deadline and as many as possible soft real-time tasks complete (or start) by their deadline. Most RTOSs are designed to be as responsive as possible to real-time tasks so when a deadline approaches, a task can be quickly scheduled.

Nonpreemptive or preemptive scheduling without priorities does not work for real-time. Priority preemptive scheduling or immediate preemption could be used instead.

2.3 Real-Time Scheduling

Scheduling approaches depend on

1. whether a system performs schedulability analysis
2. if it does, whether it is done statically or dynamically

3. whether the result of analysis produces a schedule or plan according to which tasks are dispatched at runtime

Classes of algorithms:

- **static table-driven approaches** perform a static analysis of feasible schedules of dispatching, result is a schedule that determines at runtime when a task must begin execution
  - used for periodic tasks given periodic arrival time, execution time, periodic deadline, and relative priority
  - predictable and inflexible since changes require new schedule
- **static priority-driven preemptive approaches** perform a static analysis but no schedule, instead assign priorities to tasks with a traditional priority-driven preemptive scheduler
  - priority related to time constraints
- **dynamic planning-based approaches** determine feasibility at runtime rather than statically, arriving task executed only if feasible to meet time constraints, result is schedule or plan on when to dispatch task
- **dynamic best effort approaches** have no feasibility analysis and try to meet all deadlines and abort processes whose deadline is missed
  - on arrival, a task gets a priority
  - tasks are aperiodic so no static scheduling analysis
  - unknown if deadline will be met until deadline arrives or task complete

2.4 Deadline Scheduling

Real-time applications are not concerned with speed but rather completing tasks at the right time despite dynamic resource demands and conflicts, processing overloads, and hardware or software faults.

Useful info about tasks:

- **ready time**: when task becomes ready for execution, known for period and some aperiodic tasks
- **starting deadline**: time by which a task must begin

- **completion deadline**: time by which a task must be completed
- **processing time**: time required to execute task to completion
- **resource requirements**: set of resources (other than processor) required by the task
- **priority**: relative importance of task, hard real-time tasks could have absolute priority and system failure is missed
- **subtask structure**: a task may be decomposed into a mandatory subtask (with hard deadline) and optional subtask

A policy of scheduling the task with the earliest deadline minimizes the fraction of tasks that miss their deadlines. When start deadlines are specified, use nonpreemptive scheduler (task blocks itself). When completion deadlines are specifies, use preemptive scheduler (OS preempts).

Straightforward scheme is to always schedule the ready task with earliest deadline and let it run to completion. For aperiodic tasks, this could lead to missed deadlines. Instead, earliest deadline with unforced idle times could be used where the task with the earliest deadline is scheduled and run to completion, even if processor is idle.

2.5 Rate Monotonic Scheduling

Assigns priorities to tasks on the basis of their periods, highest priority is shortest period.

Period T is the amount of time between arrival of one instance of the task and arrival of the next instance of the task. Rate is the inverse of period. Execution time C is the amount of processing time required for each occurrence where $C \leq T$ for uniprocessor. If task never denied service, then processor utilization by task is $U = C/T$.

To check if all hard deadlines are met, for n tasks to meet all deadlines, the following must hold (for a perfect scheduling algorithm):

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq 1$$

For RMS, this is

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} \leq n(2^{1/n} - 1)$$

which converges to $\ln(2) \approx 0.693$ for infinite tasks. If total utilization is less than this bound, then all tasks will be successfully scheduled.

For earliest-deadline scheduling, upper bound is 1, so more processor utilization and more periodic tasks accommodated.

Benefits of RMS:

1. performance difference is small in practice, as 0.693 is conservative
2. most hard real-time systems have soft real-time components that can execute at lower-priorities when not running RMS scheduling for hard real-time tasks
3. RMS more stable, since essential tasks must have deadlines guaranteed if schedulable, which can be done in RMS by giving them short periods or modifying RMS priorities to account for essential tasks
    (a) with earliest deadline, a periodic task's priority changes every period so less guarantees

2.6 Priority Inversion

Priority inversion occurs when circumstances within the system force a higher-priority task to wait for a lower-priority task.

- could occur if a higher-priority task attempts to lock a resource a lower-priority task already has

Unbounded priority inversion: when the duration of a priority inversion depends not only on the time required to handle a shared resource but also on the unpredictable actions of other unrelated tasks.

2.6.1 Priority Inheritance

A lower-priority task inherits the priority of any higher-priority task pending on a shared resource.

The priority change occurs as soon as the higher-priority task blocks on the resource, ends when the resource is released by the lower-priority task.

2.6.2 Priority Ceiling

A priority is associated with each resource, where the priority assigned to a resource is 1 level higher than the priority of its highest-priority user.

The scheduler dynamically assigns this priority to any task that accesses that resource, and when the task finishes with the resource, priority returns to normal.