Process Scheduling

CPU as a shared resource

- Processes in the system compete for CPU
  - Scheduler decides the process to be allocated the CPU
  - In time sharing system, many processes have to run concurrently
  - Concurrency is achieved by interleaving the processes on time share basis
    * Time quantum or time slice
    * Amount of time the process can have CPU before being evicted

- Unix scheduler
  - Works on two aspects
    1. Policy
      * Rules used to select the process to schedule next on CPU
      * Also deals with the time to switch from one process to another
      * Several conflicting objectives of policy
        - Fast response time for interactive applications
        - High throughput for background jobs
        - Avoidance of process starvation
    2. Implementation
      * Data structures and algorithms to carry out the policies
      * Policy must be implemented efficiently with minimum overhead
  - Context switch
    * Implemented as a part of scheduler
    * Kernel saves hardware execution context of current process from the u area in its PCB
    * Context contains values of general purpose, memory management, and other special registers
    * Kernel loads the hardware registers with the context of next process from the PCB of this process
    * CPU starts executing the next process from saved context
    * Expensive operation
      - Kernel must also flush data, instruction, and address translation cache to avoid incorrect memory accesses
      - New process incurs several memory accesses upon start

Clock interrupt handling

- Hardware clock interrupts the system at fixed-time intervals
  - CPU tick, clock tick, or tick
    * Time period between successive clock interrupts
    * Unix typically sets the tick to 10 ms
    * Clock frequency, or number of ticks per second, is stored in param.h as HZ
    * 10 ms tick implies a value of 100 for HZ
    * Kernel functions measure the time in number of ticks, rather than seconds or milliseconds

- Interrupt handling
Handler runs in response to hardware clock interrupt, with priority second only to power failure interrupt

Tasks of handler

- Rearm the hardware clock, if necessary
- Update CPU usage statistics for current process
- Perform scheduler-related functions
  - Priority recomputation
  - Time-slice expiration handling
- Send a SIGXCPU signal to current process if it has exceeded its CPU usage quota
- Update the time-of-day clock and other related clocks
- Handle callouts
- Wake up system processes such as swapper and pagedaemon when appropriate
- Handle alarms

All of the above tasks are not performed at every tick

Major tick

- Occurs once every n ticks
- Scheduler performs some of its tasks only on major ticks

Callouts

- Records a function to be invoked by kernel at a later time
- On Solaris, a callout is registered by `timeout(9F)`

```c
timeout_id_t timeout( void (*func)(void*), void *arg, clock_t ticks);
```

- `func` is the kernel function to invoke when the time increment expires
- `arg` is the argument to the function
- `ticks` is the number of clock ticks to wait before the function is called

Can be called from user or interrupt context

Example: In the following example, the device driver has issued an i/o request and is waiting for the device to respond. If the device does not respond within 5 seconds, the device driver will print out an error message to the console.

```c
#include <sys/types.h>
#include <sys/conf.h>

static void xxtimeout_handler ( void *arg )
{
  struct xxstate * xsp = ( struct xxstate * ) arg;
  mutex_enter ( &xsp->lock );
  cv_signal ( &xsp->cv );
  xsp->flags |= TIMED_OUT;
  mutex_exit ( &xsp->lock );
  xsp->timeout_id = 0;
}

static uint_t xxintr ( caddr_t arg )
{
  struct xxstate * xsp = ( struct xxstate * ) arg;
  ...
  ...
}
mutex_enter ( &xsp->lock );

/* Service interrupt */

_cv_signal ( &xsp->cv );
mutex_exit ( &xsp->lock );
if ( xsp->timeout_id )
{
    (void) untimeout ( xsp->timeout_id );
    xsp->timeout_id = 0;
}

return ( DDI_INTR_CLAIMED );

static void xxcheckcond ( struct xxstate * xsp )
{
    .
    .
    xsp->timeout_id = timeout ( xxtimout_handler, xsp, \
                                 ( 5 * drv_usectohz (1000000) ) );

    mutex_enter ( &xsp->lock );
    while ( /* Waiting for interrupt or timeout */ )
    {
        cv_wait ( &xsp->cv, &xsp->lock );

        if ( xsp->flags & TIMED_OUT )
            cmn_err ( CE_WARN, "Device not responding" );
    .
    .
    mutex_exit ( &xsp->lock );
    .
    .
}

– The return value from timeout(9F) is needed to cancel the callout
– Callout is cancelled by untimeout(9F)
– Callouts can be used for periodic tasks such as
  * Retransmission of network packets
  * Certain scheduler and memory management functions
  * Monitor devices to avoid losing interrupts
  * Polling devices that do not support interrupts
– Callouts are normal kernel operations and must not execute at interrupt priority
  * Clock interrupt handler does not directly invoke callouts
  * Handler checks at every tick if any callouts are due
  * If yes, it sets a flag to indicate that a callout handler must run
  * System checks the flag when it returns to base interrupt priority and if set, invokes the callout handler
  * Handler will invoke each callout that is due
  * So, callouts run only after all pending interrupts have been serviced
Kernel maintains a list of pending callouts
- List is checked on every CPU tick at high interrupt priority and so, checking time must be optimized
- Insertions into the list occur at lower priority and much less frequently than once per tick

Implementing callout list
- Sort the list in order of “time to fire”
- Kernel decrements the time of first entry at each tick and issues callout if the time reaches zero
- Another approach will be to store the absolute time and compare it with current time
- **Timing wheel**
  - Based on a hashing approach and does away with the insertion of callouts to maintain sorted order
  - Fixed-size, circular array of callout lists
  - At every tick, clock interrupt handler advances a current time pointer to the next element in the array, wrapping around at the end of array
  - Callouts on the queue are checked for time expiration
  - New callouts get inserted in the queue that is N elements or ticks away from current queue

Alarms

- Request by a process to send it a signal after a specified time
- Three types of alarms
  1. Real-time alarm
     - Signaled after actual elapsed time
     - Notified via SIGALRM signal
     - Requested by the process using
       ```c
       unsigned int alarm ( unsigned int sec );
       ```
       to send SIGALRM after sec seconds have elapsed
  2. Profiling alarm
     - Measures the amount of time the process has been executing
     - Notified via SIGPROF signal
  3. Virtual-time alarm
     - Measures the time spent by process in user mode
     - Notified via SIGVTALRM signal

- Implemented through the system calls `setitimer(2)` and `getitimer(2)`

```c
int setitimer ( int which, // Timer type
                const struct itimerval * value, // Value to set timer to
                struct itimerval * ovalue ); // Returns previous timer

int getitimer ( int which, // Timer type
                struct itimerval * value ); // Current value of timer
```

- Used to get or set the timer value for specified timer
- `setitimer(2)` returns the previous value of timer if the pointer is not set to NULL
- `itimerval` is defined as

```c
struct timeval
{
    time_t tv_sec;   // Seconds
    suseconds_t tv_usec;  // Microseconds
};
```

```c
struct itimerval
```
```c
{
    struct timeval it_interval; // Timer interval
    struct timeval it_value; // Current value
};
```

- `time_t` and `suseconds_t` are just `long`
- The value specified in `timeval` units is converted by kernel to the appropriate number of CPU ticks

- Alarms are handled only when a process is scheduled to run
  - Process priority plays an important role in determining when the alarm is handled
  - High resolution timers are useful only for high priority processes
  - Profiling and virtual time alarms may not suffer from this problem because they do not measure real time
  - The clock interrupt handler charges the entire tick to the current process even if the process uses only a part of it
  - The time measured by profiling and virtual time alarms gives the number of clock interrupts that have occurred instead of actual time
  - Averages out over long time though may be grossly inaccurate for a single alarm

### Scheduler goals

- Scheduler must be fair and deliver acceptable performance to each process
- Classifies processes based on their scheduling needs and performance expectations
  - Interactive processes
    - Spend a lot of time waiting for user inputs
    - Inputs must be processed quickly
    - Must reduce the average time and variance between user action and application response
    - For typing or mouse movement, acceptable response is 50–150ms
  - Batch processes
    - Measure of scheduling efficiency is tasks’ completion time in presence of other activity as compared to time required on an otherwise inactive system
  - Real-time processes
    - Require predictable scheduling behavior with guaranteed bounds on response time
    - Application may care more about minimizing variance than simply getting more CPU time
- Traditional schedulers work with interactive and batch processes only; real-time scheduling is provided on a system that may not run any of the interactive or batch processes

### Traditional Unix scheduling

- Traditional Unix (both SVR3 and 4.3BSD) is targeted at time-sharing, interactive environments
  - Several users run batch as well as interactive processes concurrently
  - Scheduling policy favors interactive users while preventing starvation of batch processes
- Based on priority
  - Priority of each process changes with time
  - Scheduler always selects the process with highest priority
Process Scheduling

- Preemptive time slicing for processes of equal priority
- Priority changes dynamically depending on CPU usage patterns
- A higher priority process preempts the current process even if it has not completed its time quantum
- Kernel is non-preemptible
  * Process in kernel mode cannot be preempted by a higher priority process
  * Running process can give up CPU by blocking on a resource, or when it returns from kernel mode

**Process priorities**

- Integer value between 0 and 127
- Lower number implies higher priority
- Kernel mode priorities are between 0 and 49 and user mode priorities are between 50 and 127
- Priority information in proc structure
  
  \[
  \begin{align*}
  p\_pri & \quad \text{Current scheduling priority} \\
  p\_usrpri & \quad \text{User mode priority} \\
  p\_cpu & \quad \text{Measure of recent CPU usage} \\
  p\_nice & \quad \text{User-controllable nice value}
  \end{align*}
  \]

  * p\_pri is used by scheduler to select the process to schedule
  * In user mode, p\_pri == p\_usrpri
  * If a process blocks in a system calls, and then wakes up, its priority is temporarily boosted to give preference to kernel mode processing
  * p\_usrpri holds the priority to return to from kernel mode
  * p\_pri in this case holds temporary kernel priority

- Blocked processes are assigned a sleep priority
  * Sleep priority is a kernel value and is between 0 and 49
  * Sleep priority for terminal input is 28 and for disk I/O is 20
  * When a process wakes up after blocking, kernel sets its p\_pri value to sleep priority of the event or resource
  * Lower priority numbers allow system calls to be executed promptly
    - Process may have locked some key kernel resources during system call

- Returning to user mode resets the process priority, possibly below that of another runnable process, leading to context switch

- User mode priority
  * Based on nice value and recent CPU usage
  * Nice value is a number between 0 and 39, with default being 20
  * Increasing nice value decreases the priority
  * Background processes automatically get higher nice values
  * Only superuser can decrease the nice value of a process

- Monitoring CPU usage
  * Useful in making scheduling decisions for processes
  * Derived from the field p\_cpu
    - Measure of recent CPU usage for process
    - initialized to zero upon process creation
    - Incremented by clock handler for every tick, to a maximum of 127
    - At every second, kernel invokes schedcpu() using a callout to decrease the p\_cpu value of each process by a decay factor
    - Decay factor in SVR3 is \( \frac{1}{2} \)
Decay factor in BSD is given by
\[ \text{decay} = \frac{2 \times \lambda}{2 \times \lambda + 1} \]
where \( \lambda \) is the load average, or average number of runnable process during the last second.

The user priority of each process is computed by
\[ \text{p_usrpri} = \text{PUSER} + \frac{\text{p_cpu}}{4} + 2 \times \text{p_nice} \]
where \( \text{PUSER} \) is the baseline user priority of 50.

- Process has accumulated too much CPU time
  * \( \text{p_cpu} \) factor will increase
  * Leads to a large \( \text{p_usrpri} \) value and lower priority
  * A waiting process has its \( \text{p_cpu} \) lowered by decay leading to higher priority
  * Scheme prevents starvation of a lower priority process
  * Heavily favors I/O-bound processes compared to compute-bound processes

- CPU usage factor provides for fairness and parity in scheduling time sharing processes
  * Processes move up and down in a narrow range of priorities based on their recent CPU usage
  * If priorities change too slowly, processes at lower priorities remain there for long periods leading to starvation

- Decay factor provides an exponentially weighted average of CPU usage over process’ lifetime
  * SVR3 formula
    * Simple exponential average
    * Elevates priorities when system load rises
    * Heavily loaded system gives only a small amount of time to each process
    * CPU usage value remain low
    * Decay factor reduces it even lower
    * CPU usage does not have much impact on priority
    * Lower priority processes starve
  * BDS formula
    * Decay factor depends on system load \( \lambda \)
    * High load yields small decay
    * Processes with too much CPU time lose their priority quickly

- Scheduler implementation
  - Implemented by an array of 32 queues, called \( \text{qs} \)
  - The 128 priority levels are evenly divided in these queues (4 adjacent priority levels per queue)
  - Queues are doubly linked lists, containing a pointer to the proc structures
  - A global variable \( \text{whichqs} \) contains a bitmask to indicate if there is a process in the queue
  - Only runnable processes reside in the queue
  - Selecting a process to run
    * Context switcher, \( \text{swtch()} \), selects the first queue using \( \text{whichqs} \)
    * It removes the process at the head of the queue and performs context switching
    * When \( \text{swtch()} \) returns, the newly scheduled process is dispatched
  - Context switch
    * \( \text{swtch()} \) saves the register context (general purpose registers, program counter, stack pointer, memory management registers, etc) in the PCB in the \text{u} area of the process
Then, it loads the registers from the saved context of the new process

- The \texttt{p.addr} field in the \texttt{proc} structure points to the page table entries of the \texttt{u} area and is used by \texttt{swtch} to locate the new PCB

- **Run queue manipulation**
  - Scheduler always runs the process with highest priority, unless current process is executing in kernel mode
  - The process is assigned a fixed time quantum (100ms in 4.3BSD)
  - This affects scheduling of multiple processes on the same queue
  - Every 100 milliseconds, kernel invokes \texttt{roundrobin()} through a callout to schedule the next process from the same queue
    * If a higher priority process is runnable, it is scheduled without waiting for \texttt{roundrobin()}
    * If all other runnable processes are on lower priority queues, the current process continues to run even though its quantum has expired
  - Once every second, the priority of each process is recomputed by \texttt{schedcpu()}
    * The process may end up on a different queue due to the priority recomputation
  - Every four ticks, the priority of the current process is recomputed by clock interrupt handler
  - Three situations for context switch
    1. Voluntary context switch; current process blocks on a resource or exits
    2. Priority of another process becomes more than the current one
    3. Current process, or an interrupt handler, wakes up a higher priority process
  - In voluntary switch, kernel directly calls \texttt{swtch()} from \texttt{sleep()} or \texttt{exit()}
  - Involuntary switch events occur when system is in kernel mode and hence, cannot preempt the process immediately
    * Kernel sets a flag called \texttt{runrun} to indicate that a higher priority process is waiting to be scheduled
    * When the process is about to return to user mode, kernel checks the \texttt{runrun} flag
    * If \texttt{runrun} is set, kernel transfers control to \texttt{swtch()} to initiate context switch

- **Analysis**
  - Simple and effective algorithm
  - Adequate for general time sharing with a mixture of interactive and batch jobs
  - Dynamic recomputation of priorities prevents starvation
  - Favors \texttt{i/o}-bound jobs with small infrequent \texttt{cpu} bursts
  - Scheduler limitations
    * Does not scale well for large number of processes; inefficient to recompute priorities
    * No way to guarantee a portion of \texttt{cpu} resources to a group of processes
    * No guarantees of response time to real-time applications
    * No application control over priorities; nice mechanism is not sufficient
    * Kernel is nonpreemptive resulting in a long wait for runnable high priority processes; known as \textit{priority inversion}

---

**SVR4 Scheduler**

- Improves on traditional approach due to complete redesign
- Major objectives
- Support different type of applications, including real-time applications
- Separate scheduling policy from implementation
- More control for applications over priority and scheduling
- Scheduling framework with well-defined interface to the kernel
- Allow new scheduling policies to be added in a modular manner, including dynamic loading of scheduler implementation
- Limit dispatch latency for time critical applications

- **Scheduling class**
  - Fundamental abstraction in the system
  - Defines scheduling policy for all processes in the class
  - System can provide several scheduling classes
    * Two default classes are: time sharing and real-time

- **Class-independent routines in the scheduler**
  - Implement common services such as context switching, run queue manipulation, and preemption
  - Defines the procedural interface for class-dependent functions such as priority computation and inheritance
  - Real-time class uses fixed priority
  - Time sharing class varies the priority dynamically in response to events

- **Object-oriented design**
  - Scheduler represents an abstract base class
  - Each scheduling class is a derived class

- **Class-independent layer**
  - Responsible for context switching, run queue management, and preemption
  - Highest priority process is given the CPU, except when the kernel is active; kernel stays nonpreemptible
  - Number of priorities is increased to 160 with a separate dispatch queue for each priority
  - Numerically larger values correspond to higher priorities
    * Assignment and recomputation of priorities are performed by class-dependent layer
  - Data structures for run queue management
    * `dqactmap`
      - Bitmap to show the queues with at least one runnable process
      - Processes are placed on the queue by `setfrontdq()` and `setbackdq()`, and removed by `dspdeq()`
      - The functions may be called from mainline kernel code as well as from the class-dependent routines
      - A newly runnable process is placed at the back of the queue
      - A process that is preempted before expiration of its quantum is placed at the front of the queue

- **Real-time performance**
  - Kernel is nonpreemptive, leading to problems for real-time jobs
  - Dispatch latency
    - Delay between the time when processes become runnable and when they are actually scheduled to run
    - Low value for real-time processes required
  - *Preemption points*
· Places in kernel code where kernel data structures are in stable state, and kernel is about to embark on a lengthy computation
· At such points, kernel checks a flag called kprunrun
· If set, it indicates that a real-time process is ready to run and kernel preempts the current process
· Examples of preemption points:
  Before beginning to parse each individual pathname component in `lookuppn()`
  In `open(2)`, before creating a file if it does not exist
  In memory subsystem, before freeing the pages of a process

- `runrun` flag is used, as in traditional systems, to preempt the processes about to return to user mode
- Machine-independent part of the context switch is performed by `pswitch()`, called by `swtch()`
  * After return from `pswitch()`, `swtch()` performs machine-dependent part of the context switch to manipulate register context and flush translation buffers
  * `pswitch()` performs the following functions:
    · Clear the `runrun` and `kprunrun` flags
    · Remove the process from dispatch queue
    · Update `dqactmap`
    · Set the state of the process to `SONPROC` (running on a processor)
    · Update memory management registers to map u area and virtual address translation maps of the new process

- Interface to scheduling classes
  * Generic interface with virtual functions implemented differently by each scheduling class
    * Interface defines the semantics and linkages for specific class implementations
  * `struct classfuncs`
    * Vector of pointers to functions to implement class-dependent interface
    * Global class table contains one entry for each class, containing
      · Class name
      · Pointer to an initialization function
      · Pointer to `classfuncs` vector for the class
  * Upon process creation
    * New process inherits priority class from its parent
    * Process may be moved to a different class using `priocntl(2)`
    * Scheduling classes use three field in `proc` structure
      1. `p_cid` is the class id, or an index into the global class table
      2. `p_clfuncs` is a pointer to the `classfuncs` vector for the class of the process; copied from class table entry
      3. `p_clproc` is a pointer to a class-dependent private data structure
  * Calls to generic interface are resolved through a set of macros
  * Scheduling class decides the policies for priority computation and scheduling of the processes in the class
    * Determines the range of priorities for its processes
    * Determines the conditions under which the priorities can change
    * Decides the time slice for the process each time it runs
      · Time slice may be the same for all processes, or may vary across processes depending on priority
      · Time slice can be anything from one tick to infinity
  * Entry points of class-dependent interface include:
    · `CL_TICK` is called from clock interrupt handler
· Monitors time slice
· Recomputes priority
· Handles time quantum expiration

* CL_FORK and CL_FORK_RET are called from fork(2)
  · CL_FORK initializes the child’s class-specific structures
  · CL_FORK_RET may set runrun to allow a child to run before the parent

* CL_ENTERCLASS and CL_EXITCLASS
  · Called upon entry or exit to scheduling class
  · Allocate and deallocate class-dependent data structures

* CL_SLEEP is called from sleep() and may recompute process priority

* CL_WAKEUP is called from wakeprocs()
  · Puts the process on the appropriate run queue
  · May set runrun or kprunrun

Scheduling class decides the actions for each function

* Makes scheduling versatile
  · In traditional scheduling, clock interrupt handler recomputes priority on every fourth tick
  · In new system, handler simply calls CL_TICK for the class to which the process belongs
  · For example, real-time class uses fixed priorities and does no recomputation; class-dependent code determines when the time quantum has expired and sets runrun to initiate a context switch

The 160 priorities are divide into three ranges

0-59 Time-sharing class
60-99 System priorities
100-159 Real-time class

* Time-sharing class

  Default class for a process
  · Changes process priorities dynamically
  · Uses round robin scheduling for processes with the same priority
  · Uses static dispatcher parameter table to control process priorities and time slices
  · Time slice depends on the scheduling priority
  · Parameter table defines the time slice for each priority
  · Lower the priority, larger the time slice

  Uses event-driven scheduling
  · Instead of recomputing priorities of all processes every second, changes the priority of a process in response to specific events related to the process
  · Scheduler penalizes the process by reducing its priority each time it uses up its time slice
  · Boosts the priority if the process blocks on an event or resource, or if it takes a long time to use up its quantum
  · Since only one priority is recomputed, it is fast
  · Dispatcher parameter table defines how various events change the priority of a process

Uses struct tsproc to store class-dependent data

```c
struct tsproc {
    ts_timeleft  // Time remaining in the quantum
    ts_cpupri    // System part of the priority
    ts_upri      // User part of the priority (nice value)
    ts_umdpri    // User mode priority (ts_cpupri + ts_upri, but less than 59)
    ts_dispwait  // Number of seconds of clock time since start of quantum
};
```
Process Scheduling

- Process resumes after sleeping
  - Priority of process is kernel priority, determined by sleep condition
  - Upon return to user mode, priority is restored from ts_umdpri
  - User mode priority is restricted to the range 0–59
  - ts_upri
    - Ranges from -20 to +19, with default being 0
    - Can be changed by priocntl(2) but only superuser can increase it
  - ts_cpupri is adjusted according to dispatcher parameter table
- Dispatcher parameter table
  - Present in every class (including system priorities), but is not a required structure for every class
  - Contains one entry for each priority in the class
  - For time sharing class, each entry contains the following fields
    - ts_globpri – global priority for the entry (same as index in the table)
    - ts_quantum – time quantum for the priority
    - ts_tqexp – new ts_cpupri to set when time quantum expires
    - ts_slpret – new ts_cpupri to set when returning to user mode after sleeping
    - ts_maxwait – number of seconds to wait for quantum expiry before using ts_lwait
    - ts_lwait – used in place of ts_tqexp if process took longer than ts_maxwait to use up its quantum
  - Two uses of the table
    1. Can be indexed by current ts_cpupri value to access the ts_tqexp, ts_slpret, and ts_lwait field, since these fields provide a new value of ts_cpupri based on its old value
    2. Can be indexed by ts_umdpri to access the ts_globpri, ts_quantum, and ts_maxwait fields, since these fields relate to the overall scheduling priority
- Real-time class
  - Uses priorities in the range 100–159
    - Higher priority than any time-sharing process, including those in kernel mode
    - Real-time process is scheduled before any kernel process
    - Non real-time processes in kernel mode are not preempted immediately
      - Real-time process waits until the current process returns to user mode, or reaches a kernel preemption point
    - Only superuser processes can enter the real-time class; by calling priocntl(2) and specifying the priority and time quantum
- Fixed priority and time quantum
  - Process can change these by making an explicit call to priocntl(2)
  - Real-time dispatcher parameter table is simple
    - Only stores the default quantum for each priority
    - Used if a process does not specify a quantum while entering real-time class
    - Dispatch parameter table assigns larger time slices for lower priorities
    - Class-dependent data of a real-time process is stored in struct rtproc, including the current time quantum, time remaining in the quantum, and current priority
- Processes require bounded dispatch latency as well as bounded response time
  - Both the times must have a well-defined and reasonable upper time limit
- Response time
  - Sum of time required by interrupt handler to process the event, dispatch latency, and time taken by real-time process itself to respond to the event
Traditional kernels cannot provide reasonable bound for dispatch latency since the kernel is nonpreemptible

- Process may have to wait for long time if current process is involved in elaborate kernel processing

- **Preemption points**
  - Divide lengthy kernel algorithms into smaller bounded units of work
  - When a real-time process becoming runnable
    - `rt_wakeup()` handles the class-dependent wakeup processing
    - Sets the kernel flag `kprunrun`
    - When kernel process notices the flag (at some preemption point) it initiates a context switch to the waiting real-time process
  - Wait is bounded by maximal code path between two preemption points

- **priocntl(2) system call**
  - Process scheduler control
  - Provides facilities to manipulate the priorities and scheduling behavior of a process, including a light weight process
  - The specific operations performed include
    - Changing the priority class of a process
    - Setting `ts_upri` for a time sharing process
    - Resetting priority and quantum for real-time processes
    - Obtaining current value for several scheduling parameters
  - Most of the operations are restricted to superuser
  - A variant of the call – `priocntlset(2)` provides for generalized process scheduler control over a number of processes

- **Analysis**
  - Flexible approach for addition of scheduling classes
  - Scheduler can be tailored to specific needs of applications
  - System administrator can alter the system behavior by changing the settings in dispatcher tables and rebuilding the kernel
  - Process priority is changed based on events rather than every second
  - Favors I/O-bound and interactive processes over CPU bound processes
  - Scheduling classes can be added without accessing kernel source code by the following steps:
    1. Provide an implementation of each class-dependent scheduling function
    2. initialize a `classfuncs` vector to point to these functions
    3. Provide an initialization function to perform setup tasks such as data structure allocation
    4. Add an entry for the class in a master configuration file, located in `master.d` subdirectory of kernel build directory
      * Contains pointers to the initialization function and the `classfuncs` vector
    5. Rebuild the kernel
  - In SVR4, the time-sharing class process cannot be easily switched to a different class
    - `priocntl(2)` is restricted to superuser alone
  - No provision for deadline-driven scheduling
    - Code path between preemption points may be too long for some time critical applications
  - Extremely difficult to tune system for a mixed set of applications
Solaris 2.x scheduling enhancements

- Multithreaded, symmetric-multiprocessing operating system
  - Several optimizations to lower the dispatch latency for high-priority, time-critical processes
- Preemptive kernel
  - Solaris 2.x kernel is fully preemptive (compared to preemption points of SVR4)
  - Guarantee of good response time
  - Most global kernel data structures must be protected by appropriate synchronization objects such as mutex locks or semaphores (essential requirement for a multiprocessor OS)
  - Interrupts
    * Implemented using special kernel threads
    * Threads can use standard synchronization primitives of the kernel
    * Threads block on resources if necessary
    * Solaris does not need to raise interrupt priority level to protect critical regions, and has only a few nonpreemptible code segments
    * A higher priority process can be scheduled as soon as it becomes runnable
  - Interrupt threads always run at the highest priority in the system
  - Scheduling classes can be dynamically loaded
    * Priorities of interrupt threads are recomputed to ensure that they remain at the highest possible value
    * An interrupt thread blocked on a resource must be restarted on the same processor
- Multiprocessor support
  - Single dispatch queue for all processors
  - Some threads (such as interrupt threads) may be restricted to run on a single, specific processor
  - Processors communicate with each other by sending cross-processor interrupts
  - Each processor has the following scheduling variables
    - `cpu_thread` Thread currently running on this processor
    - `cpu_dspthread` Thread last selected to run
    - `cpu_idle` Idle thread for the processor
    - `cpu_runrun` Preemption flag for time-sharing threads
    - `cpu_krunrun` Kernel preemption flag set by real-time threads
    - `cpu_chosen_level` Priority of thread that will preempt the current thread
- Hidden scheduling
  - Kernel may work asynchronously on behalf of the threads, without considering the priority of the thread for which it is doing the work
  - Exemplified by callouts
  - SVR4 hidden scheduling
    * Prior to returning a process to user level, kernel calls `runqueue()` to see if there is a pending STREAMS service request
    * Kernel processes the request by calling the service routine of the appropriate STREAMS module
    * The request is serviced by current process on behalf of a different process
    * If priority of other process is lower than the priority of current process, the request is handled at a wrong priority
    * Normal processing of current process is delayed by lower priority work
Solaris’ handling of this problem
- STREAMS processing is moved into kernel threads which run at a lower priority than any real-time thread
- Problem: Some STREAMS processing may be initiated by real-time threads
- Problem is left unresolved

Problem with callout processing
- All callouts are serviced at lowest interrupt priority which is still higher than any real-time priority
- Servicing the callout by a lower priority thread may delay a higher priority thread
- Problem is resolved by handling callouts using a callout thread running at maximum system priority, which is lower than any real-time priority
- Callouts by real-time processes are maintained separately and invoked at the lowest interrupt level, ensuring proper dispatch of time critical callouts

Priority inversion
- Lower priority process holds a resource needed by a higher priority process, blocking the higher priority process
- Problem can be solved by using priority inheritance or priority lending
  - When a higher priority thread blocks on a resource, it temporarily transfers its priority to the lower priority thread that owns the resource
- Priority inheritance must be transitive
- Solaris kernel must maintain extra state about locked objects to implement priority inheritance
  - Kernel must be able to identify the current thread owner of each locked object, and also the object for which each blocked thread is waiting
  - Since inheritance is transitive, kernel must be able to traverse all the objects and blocked threads in the synchronization chain starting from any given object