Process Management Subsystem

Process lifecycle
Scheduling
Timing
Signals and Exception handling
Coordination between other subsystems

Process memory model:

2 basic structures:
proc structure:
• Vital information required for process management subsystem
• Always in memory

user structure:
• Required when process gets CPU.
• Not addressable by the process code
• Implementation dependent parts like process HW context
• Can be swapped out when process is inactive.

Proc structure contains:
• pointer to the u structure
• current status of process (Idle, runnable, working, zombie etc.)
• Credentials (ppid, userid, group id, session id, terminal group etc.)
• Priority, current CPU usage, scheduling class information
• Wait code and data
• Signal information (signal mask, pending signals)
• Memory usage information, process address space pointer.
• Accounting information (execution time, user time, kernel time)

User structure contains:
• Register context
• stack
• Current directory
• Environment variables
• Open file descriptors table
Process states:

SIDL: Idle, proc, user created not runnable yet
SRUN: Runnable
SSLEEP: waiting for IO or sleeping
SONPROC: using processor
SZOMB: terminated
SSTOP: blocked

fork(), pswtch(), sleep(), wakeprocs(), exit(), wait()

Context switch:
- CPU service is taken from one process to the other
- Process context should be saved:
  - Sys call
  - HW trap or exception
  - sleep, IO wait
  - preemption (process completed is CPU allocation)

Clock, or Tick:
Typical value 10 milliseconds (1/100 seconds)
HZ: tick per second -> 100

HW clock interrupt is armed by the PMS to invoke a handler at each clock
- Rearm clock
- Update CPU statistics
- Scheduling computation
- Quota handling
- Update realtime clock if necessary
- handle callouts
- handle alarms
- swapper and pagedaemon scheduling

**Callouts:**

Kernel internal handler invocation.

```c
timeout(void (*fn)(), caddr_t arg, long delta)
```

execute a function after delta clocks.

Polling, scheduler events, network retransmission, implementation of user alarms etc.

Implementation:

- A sorted list of differences
- Sorted list of absolute times
- Timing wheel (circular fixed que of timeouts)

**SCHEDULER**

- Fair CPU allocation
- Good response time for interactive jobs
- Good completion time for batch and CPU intensive jobs
- Real time scheduling requirements of special tasks.

Traditional Scheduling:

multilevel feedback scheduling
processes are dynamically assigned to queues
longer running jobs are penalized
volunteer sleeping processes are favored

bitmap: to decide the highest priority with a process in the queue

```
      ....   P  P
       50     51
       ....   P
       126    P  P
       127
```
**Quantum:** number of clock ticks a process is allowed to execute in a CPU allocation.

Priority: integer in 0-127, 0-49 is reserved for system. Smaller is better.

- `p_pri`: current priority
- `p_usrpri`: user mode priority
- `p_cpu`: recent CPU usage
- `p_nice`: user controlled nice factor.

\[ \text{Pri}_i = \text{base} + \left( \frac{\text{p_cpu}_{(i-1)}}{4} \right) + 2 \times \text{p_nice} \]

\[ \text{cpu}(i)=\text{cpu}(i-1)+\left(\text{Used cpu}\right)-\text{decay factor} \]

Decay factor is either \( \text{cpu}(i-1)/2 \) or adaptive like: \( (2*\text{load_average})/(2*\text{load_average}+1) \)

**SystemV Release 4**

Priority classes:
- **Real-time**
- **System**
- **Time sharing**

Class independent layer: priority queue management, context switch, preemption
Class dependent part: calculating new priorities and assigning queues, quantum expiration etc.

time-sharing is the default class for user processes, `priocntl()` call to change class.

**System Class:**
system processes like pageout, fsflush, sched.
Always in the same priority.
60-99

**Real-time Class:**
Fixed priority. Priority is changed only by the process itself.
End of the quantum, process is inserted at the end of the priority queue of the same level.
100-159.

Dispatch parameter table:

<table>
<thead>
<tr>
<th>rt_globpri</th>
<th>rt_quantum</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>101</td>
<td>100</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>....</td>
<td>....</td>
</tr>
<tr>
<td>159</td>
<td>10</td>
</tr>
</tbody>
</table>
**Time-shared class:**

0-59.

- **ts_globpri**: default global priority
- **ts_quantum**: default time-quantum for this level
- **ts_tqexp**: assigned value to ts_cpupri when quantum expires
- **ts_slpret**: assigned value to ts_cpupri after a sleep
- **ts_dispwait**: seconds elapsed since the process started with this same quantum
- **ts_maxwait**: max. seconds that the process can keep in same quantum.
- **ts_lwait**: assigned value to ts_cpupri if ts_dispwait exceeds ts_maxwait.

**ts_dptbl[]** timesharing class priority dispatch table:

<table>
<thead>
<tr>
<th>ts_globpri</th>
<th>ts_quantum</th>
<th>ts_tqexp</th>
<th>ts_slpret</th>
<th>ts_maxwait</th>
<th>ts_lwait</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>100</td>
<td>0</td>
<td>10</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>1</td>
<td>100</td>
<td>0</td>
<td>11</td>
<td>5</td>
<td>11</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>59</td>
<td>10</td>
<td>49</td>
<td>59</td>
<td>5</td>
<td>59</td>
</tr>
</tbody>
</table>

**dispadmin -c TS -g**

- If quantum expires set priority to ts_tqexp
- If sleeps set priority to ts_slpret
- If in the same quantum for ts_maxwait seconds set priority to ts_lwait

**Class dependent Interface**

class table:

```c
struct class {
    char * cl_name; // class name
    void (*cl_init)();
    struct classfuncs *cl_funcs; // pointers to the class dependent functions
} class_t;
```

*p_cid* field, a pointer to class dependent data structures *p_clproc*, and cl_funcs entry point *p_clfuncs* is kept in proc table.

**priocntl()** system call can be used to configure classes, dispatch tables etc.

**priocntl command**: changing class characteristics of the processes

**dispadmin command**: changing configuration of classes.

**classfuncs**: both for class configuration and interface with class independent layer.
CL_ENTERCLASS, CL_FORK
CL_EXITCLASS
Executed when a process enters to and exits from a class in order to create/destroy class dependent data structures (p_clproc)
CL_FORK_RET
When fork returns there are 2 process with same exact priority characteristics. Which to run first?
CL_PREEMPT
called when a process with this class is preempted. Where to place this process in the queue? Front or back of the queue.
CL_WAKEUP
When a sleeping process awakes
CLSLEEP
When a process is set to sleep.
CL_TICK
Called at each clock cycle for the current process (SONPROC)
CL_TRAPRET
Return from a system call

Class independent layer:
• setbackq(), setfrontq() place a process on the front or back of the dispatch queue
• dispdeq() dequeue a process
• preempt() preempts the currently running process
• pswtch() selects the next highest priority process to run
• swtch() context of the currently running process is saved.

Preemption:

In order to preserve data structures of the kernel in earlier implementations, system calls were not preemptive. In order to provide real time requirements kernel preemption points are defined by two flags:
kprunrun preemption enabled for kernel
runrun preemption is enabled for a process in kernel mode
Signal handling

Proc table:
- p_sig: mask of pending signals
- p_sigmask
- p_hold: mask of signals held
- p_ignore: mask of signals ignored
- p_siginfo: mask of signals require more information (sigaction)
- p_sigqueue: pointer to the queue of siginfo structures
- p_cursig: most current signal being handled

User structure:
Alternate stack information, handler reset flags, system call restarting information
- u_sigmas[MAXSIG]: when a signal is caught mask of held signals
- u_signal[MAXSIG]: signal handling dispositions

Process management system decides what to do in case of a signal is generated.

System Call Traps

system call information is kept in:
struct sysent sysent[NSYSENT] table.
struct sysent {
    char sy_narg; // Number of arguments
    char sy_flags;
    int (*sy_call)(); // system call service handler function
};

systrap() function executes a trap instruction for entering kernel mode.
Each system call has an integer id and corresponding action in sysent is taken.
Arguments are kept in user structure field u_args where number of arguments is in the sy_narg field.

Kernel text
1  open()
2     close()
3     pipe()
4     fork()
5     exec()
......

Return value and arguments are passed and returned depends on the implementation.