Mach Scheduling

Designed for MultiProcessors

Timesharing Environment

Provision for Other Scheduling Policies

Four Components:

- Thread State Machine
- Run Queues
- Thread Priorities
- Processor Allocation

Thread State Machine

State composed of 4 orthogonal components:

- Run
  Thread running, or on run queue
- Wait
  Thread waiting, or queued to wait on an event
- Uninterruptible
  Thread's wait is not interruptible
- Suspend
  Thread is suspended, or should become suspended at next interruptible wait

Not all combinations exist.

State Combinations

R-Proc	Running
RS-Proc	Running, suspend pending
R[N]-Runq	On run queue
RS[N]-Runq	On run queue, suspend pending
RW[N]	Running, wait pending
RWS[N]	Running, wait pending, suspend pending
S	Suspended
W[N]	Waiting
WS	Waiting and suspended
WSN	Waiting, suspend pending, uninterruptible

Transitions

Assert: Declare event to be waited for

Run -> Run + Wait

Wakeup: Wake up threads waiting for an event
Clear: Wake up a specific thread

Wait -> Run-Runq

Run + Wait -> Run-Proc

Wait + Suspend -> Suspend

Block: Context switch - pending suspends and waits occur

Run + Wait -> Wait

Run + Suspend -> Suspend (if interruptible)

Run + Wait + Suspend -> Wait + Suspend (if interruptible)

Run -> Run-Runq

Choose: Chooses next thread to run and runs it

Run-Runq -> Run-Proc

Marks thread Interruptible

Hold: Request suspension

Run -> Run + Suspend (request)
Wait -> Wait + Suspend

Release: Release thread suspension

Wait + Suspend -> Wait

Suspend -> Run-Runq

Dowait: Wait for pending suspend

Waits until state is Suspend or Wait + Suspend

"Swap" state (whether thread has a kernel stack) is independent of scheduling state.

- Thread has kernel stack
- Thread has no kernel stack
- Thread is queued for "swapin" thread (no kernel stacks available)

Both states are in same word so that stack handoff operations can quickly check for waiting thread with no stack

Run Queues

Run Queue Structure:

- 32 Queues (0 - highest priority)
- Count
- Hint for Highest Priority Queue

Global Run Queue - Processor Set

Local Run Queue - Processor

Local run queue is only used for threads bound to one processor. Used by unparallelized device drivers, processor allocation

Thread_Setrun - Enters thread on run queues

- Bound Processor
- Last Processor if idle
- Idle Processor
- Processor Set

Choose-Thread - Finds thread on run queues

- Local Run Queue
- Global Run Queue

Locking:

Thread_setrun locks thread, then run queue.

Choose_thread locks run queue.

Rem_runq (used by thread_dowait) locks thread; if thread->runq is not NULL, then it locks runq.

Idle Thread

Runs when no other thread to run

Processor in Idle Queue

Next thread to run is in *processor->next_thread*, not in run queue - fast exit from idle state

Avoids conflicting use of last thread’s kernel stack on multiprocessor

Thread Priorities

Thread Priorities 0 - 31

Current Priority is Base Priority + CPU Usage

Priority drops as thread runs

Priority rises as thread waits

Fixed-priority threads have no usage component

Priority Aging

CPU time (user + system time, in microseconds) is accumulated when thread runs

Scheduling usage is multiplied by load factor, to compensate for heavily loaded CPUS. Ensures that thread priorities don't rise solely due to increased load.

Load Factor =
min(number of threads / number of CPUS, 1)

Usage aging:

usage = old usage * (5/8) + new usage

Load Factor Aging

load = (old_load + new_load) / 2

New usage and priority computed once per second

Thread computes usage and lowers priority while it runs

Thread_setrun updates usage for thread being awakened

Sched_thread updates usage for low-priority threads that are stuck on run queues

Each second of averaged, weighted CPU usage is equivalent to 3.8 priority levels

Thread Quantum

Controls rate of context switches: minimum quantum is 100 ms

For timesharing, thread will not be pre-empted until its first quantum has expired

For fixed-priority, thread will not switch to another of same priority until its first quantum has expired

For multiprocessors:

if number of threads < twice the number of processors, quantum is extended to prevent rapid context switching

Extended quanta are distributed among processors to avoid simultaneous context switching

Processor Allocation

Processors allocated to Processor Sets

Threads bound to Processor Sets will run on any processor in that set

Any User can create a processor set

Privileges needed to allocate processors to processor set

Default processor set always has at least one processor

To keep locking out of critical path in choose_thread, a processor must move itself from one processor set to another

Action Thread

Binds itself to processor to change processor sets or to shut down processor

If last processor in processor set, must suspend all threads assigned to processor set