CPU Scheduling Basic Concepts (1)

- **Maximum CPU utilization** is obtained with multiprogramming since another process can be assigned to use the CPU while other processes are waiting for something.
  - without multiprogramming the CPU sits idle while a process waits
- **CPU Burst** - time that a process is expected to be executing without requesting a blocking operation (I/O, wait, etc)
- **I/O Burst** - time that a process is expected to spend in an I/O operation

CPU Scheduling Basic Concepts (2)

- **CPU–I/O Burst Cycle** – Process execution consists of a cycle of CPU execution and I/O wait
- **CPU burst** distribution is generally characterized as exponential, as illustrated on the next figure (next slide).
  - there is usually a large number of short CPU bursts and a small number of short CPU bursts.

Process Behavior (1)

Figure 2-22. Bursts of CPU usage alternate with periods of waiting for I/O.
(a) A CPU-bound process. (b) An I/O-bound process.

Scheduling Algorithms

**All systems**
- Fairness — giving each process a fair share of the CPU
- Policy enforcement — seeing that stated policy is carried out
- Balance — keeping all parts of the system busy

**Batch systems**
- Throughput — maximize jobs per hour
- Turnaround time — minimize time between submission and termination
- CPU utilization — keep the CPU busy all the time

**Interactive systems**
- Response time — respond to requests quickly
- Proportionality — meet users’ expectations

**Real—time systems**
- Meeting deadlines — avoid losing data
- Predictability — avoid quality degradation in multimedia systems

Figure 2-23. Some goals of the scheduling algorithm under different circumstances.
When to Schedule (1)

- When scheduling is absolutely required:
  1. When a process exits.
  2. When a process blocks.
- When scheduling usually done (though not absolutely required)
  3. When a new process is created.
  4. When an I/O interrupt occurs.
  5. When a clock interrupt occurs.

When to Schedule (2)

- 1, 2, 3 are nonpreemptive
  - a process can not be removed from running state unless it finishes or voluntarily request being moved to ready state
- 4, 5 are preemptive
  - a process can be forced out of running state at any moment

Dispatcher

- **Dispatcher module** gives control of the CPU to the process selected by the short-term scheduler; this involves:
  - switching context
  - switching to user mode
  - jumping to the proper location in the user program to restart that program
- **Dispatch latency** – time it takes for the dispatcher to stop one process and start another running

Scheduling Criteria

- **CPU utilization** – keep the CPU as busy as possible
- **Throughput** – # of processes that complete their execution per time unit
- **Turnaround time** – amount of time to execute a particular process
- **Waiting time** – amount of time a process has been waiting in the ready queue
- **Response time** – amount of time it takes from when a request was submitted until the first response is produced, not output (for time-sharing environment)
Scheduling Optimization Criteria

- Maximize CPU utilization
- Maximize throughput
- Minimize turnaround time
- Minimize waiting time
- Minimize response time

First-Come, First-Served (FCFS) Scheduling

**FCFS Scheduling** - processes are served in the order of arrival. Consider the following example:

Consider three processes with given burst times.

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

Suppose that the processes arrive in the order: P₁, P₂, P₃
The Gantt Chart for the schedule is:

\[
\begin{array}{c|c|c|c}
    & P₁ & P₂ & P₃ \\
0    &    &    &   \\
30   &    &    &   \\
\end{array}
\]

Waiting time for P₁ = 0; P₂ = 24; P₃ = 27
Average waiting time: \((0 + 24 + 27)/3 = 17\)

Convoy effect - short process behind long process

Shortest-Job-First (SJF) Scheduling

- Associate with each process the length of its next CPU burst. Use these lengths to schedule the process with the shortest time
- SJF is optimal – gives minimum average waiting time for a given set of processes
  - The difficulty is knowing the length of the next CPU burst

FCFS Scheduling (Cont)

Suppose that the processes arrive in the order P₂, P₃, P₁
The Gantt chart for the schedule is:

\[
\begin{array}{c|c|c|c}
    & P₂ & P₃ & P₁ \\
0    &    &    &   \\
6     &    &    &   \\
30    &    &    &   \\
\end{array}
\]

Waiting time for P₁ = 6; P₂ = 0; P₃ = 3
Average waiting time: \((6 + 0 + 3)/3 = 3\)
Much better than previous case
Convoy effect - short process behind long process
Example of SJF

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>6</td>
</tr>
<tr>
<td>P₂</td>
<td>8</td>
</tr>
<tr>
<td>P₃</td>
<td>7</td>
</tr>
<tr>
<td>P₄</td>
<td>3</td>
</tr>
</tbody>
</table>

Average waiting time = (3 + 16 + 9 + 0) / 4 = 7

SJF scheduling chart

Example of SRJF

Consider the following 4 process with the arrival/burst times shown

<table>
<thead>
<tr>
<th>Process</th>
<th>Arrival Time</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>0.0</td>
<td>8</td>
</tr>
<tr>
<td>P₂</td>
<td>1.0</td>
<td>4</td>
</tr>
<tr>
<td>P₃</td>
<td>2.0</td>
<td>9</td>
</tr>
<tr>
<td>P₄</td>
<td>3.0</td>
<td>5</td>
</tr>
</tbody>
</table>

SRJF scheduling chart

Average waiting time = (10-1)+(1-1)+(17-2)+(5-3) / 4 = 6.5

Note that process 1 is preempted at time 1, then it waits up to time 10, etc...

Shortest Remaining Job First

- An extension to SRJF is to preempt the running process whenever a new process arrives with burst time that is smaller than the remaining time of the one in Running state.

Determining Length of Next CPU Burst

- The major problem of SJF-based scheduling is how to know the next CPU burst of processes.
- Can only estimate it by using statistical strategies.
- Can be done by using the length of previous CPU bursts, using exponential averaging - averages the past bursts to estimate the next.

1. \( t_n \) = actual length of \( n^{th} \) CPU burst
2. \( \tau_{n+1} \) = predicted value for the next CPU burst
3. \( \alpha, 0 \leq \alpha \leq 1 \) is the weight factor
4. Define: \( \tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n \)
**Prediction of the Length of the Next CPU Burst**

The figure shows an exponential average with $\alpha = 1/2$ and $\tau = 10$.

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \tau_n$$

**Examples of Exponential Averaging**

- $\alpha = 0$
  - $\tau_{n+1} = \tau_n$
  - Recent history does not count
- $\alpha = 1$
  - $\tau_{n+1} = \alpha t_n$
  - Only the actual last CPU burst counts

If we expand the formula, we get:

$$\tau_{n+1} = \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \ldots + (1 - \alpha)^j \alpha t_{n-j} + \ldots + (1 - \alpha)^{n+1} \tau_0$$

Since both $\alpha$ and $(1 - \alpha)$ are less than or equal to 1, each successive term has less weight than its predecessor.

**Priority Scheduling**

- A priority number (integer) is associated with each process
- The CPU is allocated to the process with the highest priority (smallest integer = highest priority)
  - Preemptive
  - Nonpreemptive
- SJF is a priority scheduling where priority is the predicted next CPU burst time
- Problem: Starvation – low priority processes may never execute
- Solution: Aging – as time progresses increase the priority of the process

Figure 2-27. A scheduling algorithm with four priority classes.
Priority Scheduling

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>10</td>
<td>3</td>
</tr>
<tr>
<td>P₂</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>P₃</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>P₄</td>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>P₅</td>
<td>5</td>
<td>2</td>
</tr>
</tbody>
</table>

- Priority Scheduling (Gantt chart)
- Average waiting time = (6 + 0 + 16 + 19 + 1)/5 = 8.2

Round Robin (RR)

- Each process gets a small unit of CPU time (time quantum), usually 10-100 milliseconds. After this time has elapsed, the process is preemted and added to the end of the ready queue.
- If there are \( n \) processes in the ready queue and the time quantum is \( q \), then each process gets \( 1/n \) of the CPU time in chunks of at most \( q \) time units at once. No process waits more than \( (n-1)q \) time units.
- Performance
  - \( q \) large \( \Rightarrow \) FIFO
  - \( q \) small \( \Rightarrow \) increases context switch
  - \( q \) must be large with respect to context switch, otherwise overhead is too high

Example of RR with Time Quantum = 4

<table>
<thead>
<tr>
<th>Process</th>
<th>Burst Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>P₁</td>
<td>24</td>
</tr>
<tr>
<td>P₂</td>
<td>3</td>
</tr>
<tr>
<td>P₃</td>
<td>3</td>
</tr>
</tbody>
</table>

The Gantt chart is:

Typically, higher average turnaround than SJF, but better response
Most modern systems have time quanta in range **10-100 msec.**

- Context switch is typically less than 10 microseconds.

### Average Turnaround Time Varies With Quantum Length (1)

- Att can be improved if most processes finish their next CPU burst in a single time quantum.
- Example: Consider 3 process with CPU burst of 10 each.
  - q=1 => att = 29
  - q=10 => att = 20
- If context switch time is added, att increases more as q becomes smaller.

### Turnaround Time Varies With Quantum Length (2)

- Time q should be large compared to cs time, but not too large.
  - If too large, RR degenerates to FCFS.
  - Rule of thumb is that **80 percent of the CPU bursts should be shorter than the time q.**
Multilevel Queue

- Ready queue is partitioned into separate queues:
  - foreground (interactive)
  - background (batch)
- Each queue has its own scheduling algorithm
  - foreground – RR
  - background – FCFS
- Scheduling must be done between the queues
  - Fixed priority scheduling; (i.e., serve all from foreground then from background). Possibility of starvation.
  - Time slice – each queue gets a certain amount of CPU time which it can schedule amongst its processes; i.e., 80% to foreground in RR, 20% to background in FCFS

Multilevel Feedback Queue

- A process can move between the queues; Ex: aging can be implemented this way
- Scheduler defined by the following:
  - number of queues
  - scheduling algorithms for each queue
  - method used to determine when to upgrade a process
  - method used to determine when to demote a process
  - method used to determine which queue a process will enter when it needs service

Example of Multilevel Feedback Queue

- Three queues:
  - $Q_0$ – RR with time quantum 8 milliseconds
  - $Q_1$ – RR time quantum 16 milliseconds
  - $Q_2$ – FCFS
- Scheduling
  - A new job enters $Q_0$ which is served FCFS. When it gains CPU, job receives 8 ms. If it does not finish, job is moved to $Q_1$.
  - At $Q_1$ job is again served FCFS and receives 16 ms. If it still does not complete, it is moved to queue $Q_2$. 
Multilevel Feedback Queues

- **Hard real-time** systems – required to complete a critical task within a guaranteed amount of time
- **Soft real-time** computing – requires that critical processes receive priority over less fortunate ones

Real-Time Scheduling

- **Thread Scheduling (1)**
  - Distinction between user-level and kernel-level threads
    - On many-to-one and many-to-many models, user level threads are scheduled by the thread libraries, whereas kernel threads are scheduled by the kernel.
    - The thread library schedules user-level threads to run on available LWPs.
    - The scheme is known as **process-contention scope (PCS)** since scheduling competition is within the process

  **Thread Scheduling (2)**
  - Kernel threads are scheduled onto available CPU based on **system-contention scope (SCS)** – competition among all threads in system
    - this includes those systems that implement the one-to-tone model between user and kernel threads...

PCS is based on priorities of threads. Thread in CPU is preempted if a higher priority thread arrives.
- No guarantee of time slice among threads of equal priority.
Thread Scheduling (PCS)

Figure 2-28. (a) Possible scheduling of user-level threads with a 50-msec process quantum and threads that run 5 msec per CPU burst.

Thread Scheduling (SCS)

Figure 2-28. (b) Possible scheduling of kernel-level threads with the same characteristics as (a).

PTHREAD Scheduling

- API allows specifying either PCS or SCS during thread creation
  - PTHREAD SCOPE PROCESS schedules threads using PCS scheduling
  - PTHREAD SCOPE SYSTEM schedules threads using SCS scheduling.

PTHREAD Scheduling API

```c
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5
/* Each thread will begin control in this function */
void *runner(void *param) {
    printf("I am a thread\n");
    pthread_exit(0);
}

int main(int argc, char *argv[]) {  
    int i;
    pthread_t tid[NUM THREADS];
    pthread_attr_t attr;
    /* get the default attributes */
    pthread_attr_init(&attr);
    /* set the scheduling algorithm to PROCESS or SYSTEM */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
    /* set the scheduling policy - FIFO, RT, or OTHER */
    pthread_attr_setschedpolicy(&attr, SCHED_OTHER);
    /* create the threads */
    for (i = 0; i < NUM THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);
    /* now join on each thread */
    for (i = 0; i < NUM THREADS; i++)
        pthread_join(tid[i], NULL);
    return 0;
}
```
Three Level Scheduling (1)

Criteria for deciding which process to choose:
- How long has it been since the process was swapped in or out?
- How much CPU time has the process had recently?
- How big is the process? (Small ones do not get in the way.)
- How important is the process?

Multiple-Processor Scheduling (1)
- Scheduling problem is more complex if multiple CPUs are available
- We are using a model of **homogeneous processors** within a multiprocessor
  - all processors have equal functionality in terms of hardware design

Multiple-Processor Scheduling (2)
- **Asymmetric multiprocessing**
  - one CPU (master) handles all kernel operations
  - other processors (slaves) execute processes assigned by master
  - kernel operations are in general as in a single CPU system
- **Symmetric multiprocessing (SMP)** — each processor is self-scheduling, all processes in shared ready queue, or each has its own private queue of ready processes
Processor Affinity

- **Processor affinity** – process has affinity for processor on which it is currently running. For example: there may be benefits in cached data items - some may still be there from previous CPU bursts of the process...
  - **soft affinity** - whenever the system tries to achieve processor affinity but cannot guarantee it
  - **hard affinity** - some systems (e.g. Linux) provide system calls that allow a process to establish that it is not to migrate to other processors...

Multicore Processors

- Recent trend to place multiple processor cores on same physical chip
- Faster and consume less power
- Multiple threads per core also growing
  - Takes advantage of memory stall to make progress on another thread while memory retrieve happens

NUMA and CPU Scheduling

Multithreaded Multicore System

Another thread can be assigned to core during this time. This requires support from hardware to provide more than one "hardware thread".
Operating System Examples

- Solaris
- Windows XP
- Linux
- Java
- MINIX

Solaris Scheduling (1)

- Priority-based thread scheduling:
  - A thread with a given priority has a chance only if no thread of higher priority is not waiting for CPU
- Each thread belongs to one of six classes:
  - time sharing, interactive, real time, systems, fair share, and fixed priority
- Priorities are numbers: 0-169 (global priority)
- The higher the number the higher the priority

Solaris Scheduling (2)

- Each class has a range of possible priorities
- Priorities of threads in time-sharing and interactive classes may change as they execute. The table on the next slide shows how these transitions are determined for some of the priorities on those two classes...
  - The table also shows the quantum assigned to each priority.
  - Lower priorities are assigned larger quantum

Solaris Dispatch Table

```
<table>
<thead>
<tr>
<th>priority</th>
<th>time quantum</th>
<th>time quantum expired</th>
<th>return from sleep</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>5</td>
<td>200</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>10</td>
<td>160</td>
<td>0</td>
<td>51</td>
</tr>
<tr>
<td>15</td>
<td>160</td>
<td>5</td>
<td>51</td>
</tr>
<tr>
<td>20</td>
<td>120</td>
<td>10</td>
<td>52</td>
</tr>
<tr>
<td>25</td>
<td>120</td>
<td>15</td>
<td>52</td>
</tr>
<tr>
<td>30</td>
<td>80</td>
<td>20</td>
<td>53</td>
</tr>
<tr>
<td>35</td>
<td>80</td>
<td>25</td>
<td>54</td>
</tr>
<tr>
<td>40</td>
<td>40</td>
<td>30</td>
<td>55</td>
</tr>
<tr>
<td>45</td>
<td>40</td>
<td>35</td>
<td>56</td>
</tr>
<tr>
<td>50</td>
<td>40</td>
<td>40</td>
<td>58</td>
</tr>
<tr>
<td>55</td>
<td>40</td>
<td>45</td>
<td>58</td>
</tr>
<tr>
<td>59</td>
<td>20</td>
<td>49</td>
<td>59</td>
</tr>
</tbody>
</table>
```

Priorities of threads in time-sharing and interactive classes may change as they execute.
Linux Scheduling (1)

- Linux uses a preemptive, priority-based, algorithm.
- Constant order $O(1)$ scheduling time
- Two priority ranges: time-sharing and real-time
- **Real-time** range from 0 to 99 and **nice** value from 100 to 140
  - the lower the value, the higher the priority.

Linux Scheduling (2)

- Figure shows relationship between priorities and quantum.
- A runnable task is considered eligible for execution on the CPU as long as it has time left in its time-slice.
Two array Task Indexing

- When its time-slice has been exhausted, if task is still runnable, it passes to expired array.
- When active array has no task, references to both arrays are swapped - expired becomes active

Algorithm Evaluation (1)

- How do we select a CPU scheduling algorithm for a particular system?
- First, define the criteria (e.g. minimize average waiting time)
- Deterministic modeling
  - Given a predetermined workload is known
  - Guestimate the performance of each algorithm (e.g. SJF, FCFS, RR)
  - Useful if behavior repeats or the same workload repeats

Queueing models

- Deterministic model is not realistic

Simulation

- Use random number generator to generate processes
- Use traces

Implementation

Algorithm Evaluation (2)

- Queueing models
  - Deterministic model is not realistic

Queueing Theory

- Arriving requests enter the queue
- Wait until service provided
Queueing Theory Example: Storage Area Network (SAN) System

- Little's Theorem: For a system in steady state:
  \[ Q_L = \lambda T_Q \]

Queue Descriptors (1)
- Generic descriptor: A/S/m/k
  - A denotes the arrival process
    - For Poisson arrivals we use M (for Markovian)
  - B denotes the service-time distribution
    - M: exponential distribution
    - D: deterministic service times
    - G: general distribution

Queue Descriptors: Examples
- M/M/1: Poisson arrivals, exponentially distributed service times, one server, infinite buffer
- M/M/m: same as previous with m servers
- M/M/m/m: Poisson arrivals, exponentially distributed service times, m server, no buffering
- M/G/1: Poisson arrivals, identically distributed service times follows a general distribution, one server, infinite buffer
- */D/∞: A constant delay system

Queue Descriptors (2)
- m is the number of servers
- k is the max number of customers allowed in the system – either in the buffer or in service
  - k is omitted when the buffer size is infinite
Evaluation of CPU schedulers by Simulation

Java Thread Scheduling

- JVM Uses a Preemptive, Priority-Based Scheduling Algorithm
- FIFO Queue is Used if There Are Multiple Threads With the Same Priority

Java Thread Scheduling (cont)

- JVM Schedules a Thread to Run When:
  - The Currently Running Thread Exits the Runnable State
  - A Higher Priority Thread Enters the Runnable State

- * Note – the JVM Does Not Specify Whether Threads are Time-Sliced or Not

Time-Slicing

- Since the JVM Doesn’t Ensure Time-Slicing,
- The `yield()` Method May Be Used:

  ```java
  while (true) {
    // perform CPU-intensive task
    ...
    Thread.yield();
  }
  ```

- This Yields Control to Another Thread of Equal Priority
### Thread Priorities

<table>
<thead>
<tr>
<th>Priority</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread.MIN_PRIORITY</td>
<td>Minimum Thread Priority</td>
</tr>
<tr>
<td>Thread.MAX_PRIORITY</td>
<td>Maximum Thread Priority</td>
</tr>
<tr>
<td>Thread.NORM_PRIORITY</td>
<td>Default Thread Priority</td>
</tr>
</tbody>
</table>

Priorities may be set using `setPriority()` method:
```
setPriority(Thread.NORM_PRIORITY + 2);
```

### Thread Priorities

Scheme that may be followed by a threads scheduler in Java. This presumes a single CPU system.
- `tQueue` contains low priority threads to be managed...
- `tQueue.front()` is the thread to be using the CPU
- the scheduler runs at max priority.

The following shows part of the scheduler thread's work:
```
while (!(tQueue.isEmpty())) {
    // Scheduler goes to sleep...
    Thread.sleep(QUANTUM);
    ...
    // Time to change... after scheduler wakes up.
    if (tQueue.front().isAlive()) {
        tQueue.front().setPriority(Thread.MIN_PRIORITY);
        if (!tQueue.dequeue())
            tQueue.enqueue(tQueue.dequeue());
    } else
        tQueue.dequeue();
    if (!tQueue.isEmpty())
        tQueue.front().setPriority(Thread.NORM_PRIORITY);
}
```

### Solaris 2 Scheduling

The Internal Structure of MINIX

![Diagram of Solaris 2 Scheduling](image)

![Diagram of The Internal Structure of MINIX](image)

Figure 2-29. MINIX 3 is structured in four layers. Only processes in the bottom layer may use privileged (kernel mode) instructions.
Sizes of Types in MINIX

<table>
<thead>
<tr>
<th>Type</th>
<th>16-Bit MINIX</th>
<th>32-Bit MINIX</th>
</tr>
</thead>
<tbody>
<tr>
<td>gid_t</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>dev_t</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>pid_t</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>ino_t</td>
<td>16</td>
<td>32</td>
</tr>
</tbody>
</table>

Figure 2-33. The size, in bits, of some types on 16-bit and 32-bit systems.

MINIX Message Types

Figure 2-34. The seven message types used in MINIX 3. The sizes of message elements will vary, depending upon the architecture of the machine; this diagram illustrates sizes on CPUs with 32-bit pointers, such as those of Pentium family members.

MINIX 3 Startup

<table>
<thead>
<tr>
<th>Component</th>
<th>Description</th>
<th>Loaded by</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel</td>
<td>Kernel + clock and system tasks</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>pm</td>
<td>Process manager</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>fs</td>
<td>File system</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>rs</td>
<td>(Re)starts servers and drivers</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>memory</td>
<td>RAM disk driver</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>log</td>
<td>Buffers log output</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>tty</td>
<td>Console and keyboard driver</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>driver</td>
<td>Disk (at, blop, or floppy) driver</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>init</td>
<td>Parent of all user processes</td>
<td>(in boot image)</td>
</tr>
<tr>
<td>floppy</td>
<td>Floppy driver (if booted from hard disk) /etc/rc</td>
<td></td>
</tr>
<tr>
<td>is</td>
<td>Information server (for debug dumps) /etc/rc</td>
<td></td>
</tr>
<tr>
<td>cmos</td>
<td>Reads CMOS clock to set time /etc/rc</td>
<td></td>
</tr>
<tr>
<td>random</td>
<td>Random number generator /etc/rc</td>
<td></td>
</tr>
<tr>
<td>printer</td>
<td>Printer driver /etc/rc</td>
<td></td>
</tr>
</tbody>
</table>

Figure 2-30. Some important MINIX 3 system components. Others such as an Ethernet driver and the inet server may also be present.

Scheduling in MINIX

Figure 2-43. The scheduler maintains sixteen queues, one per priority level. Shown here is the initial queuing process as MINIX 3 starts up.
Hardware-Dependent Kernel Support

![Segment Descriptor](image)

Figure 2-44. The format of an Intel segment descriptor.

Overview of System Task (1)

<table>
<thead>
<tr>
<th>Message type</th>
<th>From</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_fork</td>
<td>PM</td>
<td>A process has forked</td>
</tr>
<tr>
<td>sys_exec</td>
<td>PM</td>
<td>Set stack pointer after EXEC call</td>
</tr>
<tr>
<td>sys_exit</td>
<td>PM</td>
<td>A process has exited</td>
</tr>
<tr>
<td>sys_nice</td>
<td>PM</td>
<td>Set scheduling priority</td>
</tr>
<tr>
<td>sys_priexit</td>
<td>RS</td>
<td>Set or change privileges</td>
</tr>
<tr>
<td>sys_trace</td>
<td>PM</td>
<td>Carry out an operation of the PTRACE call</td>
</tr>
<tr>
<td>sys_kill</td>
<td>PM, FS, TTY</td>
<td>Send signal to a process after KILL call</td>
</tr>
<tr>
<td>sys_getmsg</td>
<td>PM</td>
<td>PM is checking for pending signals</td>
</tr>
<tr>
<td>sys_endmsg</td>
<td>PM</td>
<td>PM has finished processing signal</td>
</tr>
<tr>
<td>sys_sigprocm</td>
<td>PM</td>
<td>Send a signal to a process</td>
</tr>
<tr>
<td>sys_sigtell</td>
<td>PM</td>
<td>Cleanup after completion of a signal</td>
</tr>
<tr>
<td>sys_irqtell</td>
<td>Drivers</td>
<td>Enable, disable, or configure interrupt</td>
</tr>
</tbody>
</table>

Figure 2-45. The message types accepted by the system task. “Any” means any system process; user processes cannot call the system task directly.

Overview of System Task (2)

<table>
<thead>
<tr>
<th>Message type</th>
<th>From</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>sys_devio</td>
<td>Drivers</td>
<td>Read from or write to an I/O port</td>
</tr>
<tr>
<td>sys_devio</td>
<td>Drivers</td>
<td>Read or write string from/to I/O port</td>
</tr>
<tr>
<td>sys_vdevio</td>
<td>Drivers</td>
<td>Carry out a vector of I/O requests</td>
</tr>
<tr>
<td>sys_memop</td>
<td>Drivers</td>
<td>Do a real-mode BIOS call</td>
</tr>
<tr>
<td>sys_newmsg</td>
<td>PM</td>
<td>Set up a process memory map</td>
</tr>
<tr>
<td>sys_userop</td>
<td>Drivers</td>
<td>Add segment and get selector (far data access)</td>
</tr>
<tr>
<td>sys_memop</td>
<td>PM</td>
<td>Write char to memory area</td>
</tr>
<tr>
<td>sys_vmemop</td>
<td>Drivers</td>
<td>Convert virtual address to physical address</td>
</tr>
<tr>
<td>sys_vcopy</td>
<td>FS, Drivers</td>
<td>Copy using pure virtual addressing</td>
</tr>
<tr>
<td>sys_physcopy</td>
<td>Drivers</td>
<td>Copy using physical addressing</td>
</tr>
<tr>
<td>sys_vcopy</td>
<td>Any</td>
<td>Vector of VCOPY requests</td>
</tr>
<tr>
<td>sys_physio</td>
<td>Any</td>
<td>Vector of PHYSIOCOPY requests</td>
</tr>
<tr>
<td>sys_times</td>
<td>PM</td>
<td>Get uptime and process times</td>
</tr>
<tr>
<td>sys_setalarm</td>
<td>PM, FS, Drivers</td>
<td>Schedule a synchronous alarm</td>
</tr>
<tr>
<td>sys_abort</td>
<td>PM, TTY</td>
<td>Panic: MINIX is unable to continue</td>
</tr>
<tr>
<td>sys_getinfo</td>
<td>Any</td>
<td>Request system information</td>
</tr>
</tbody>
</table>

Figure 2-45. The message types accepted by the system task. “Any” means any system process; user processes cannot call the system task directly.

The Clock Task in MINIX 3

![Clock Task Diagram](image)

(a) Worst case for reading a block requires seven messages. (b) Best case for reading a block requires four messages.
Clock Hardware

![A programmable clock diagram](image)

Figure 4-47. A programmable clock.

Clock Software (1)

Typical duties of a clock driver.
1. Maintain time of day
2. Prevent processes from running longer than allowed
3. Accounting for CPU usage
4. Handling alarm system call by user processes
5. Providing watchdog timers for parts of system itself
6. Doing profiling, monitoring, and statistics gathering

Clock Software (2)

![Three ways to maintain the time of day](image)

Figure 2-48. Three ways to maintain the time of day.

Clock Software (3)

![Simulating multiple timers with a single clock](image)

Figure 2-49. Simulating multiple timers with a single clock.
Summary of Clock Services

<table>
<thead>
<tr>
<th>Service</th>
<th>Access</th>
<th>Response</th>
<th>Clients</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_uptime</td>
<td>Function call</td>
<td>Ticks</td>
<td>Kernel or system task</td>
</tr>
<tr>
<td>set_timer</td>
<td>Function call</td>
<td>None</td>
<td>Kernel or system task</td>
</tr>
<tr>
<td>reset_timer</td>
<td>Function call</td>
<td>None</td>
<td>Kernel or system task</td>
</tr>
<tr>
<td>read_clock</td>
<td>Function call</td>
<td>Count</td>
<td>Kernel or system task</td>
</tr>
<tr>
<td>clock_stop</td>
<td>Function call</td>
<td>None</td>
<td>Kernel or system task</td>
</tr>
<tr>
<td>Synchronous alarm</td>
<td>System call</td>
<td>Notification</td>
<td>Server or driver, via system task</td>
</tr>
<tr>
<td>POSIX alarm</td>
<td>System call</td>
<td>Signal</td>
<td>User process, via PM</td>
</tr>
<tr>
<td>Time</td>
<td>System call</td>
<td>Message</td>
<td>Any process, via PM</td>
</tr>
</tbody>
</table>

Figure 2-50. The time-related services supported by the clock driver.