Kernel Organization

- The normal way to enter the kernel is through system calls. Most of the code in the kernel is executed by processes that have made a system call.
- Some services are performed by special *kernel processes*. These processes have their own *process struct*, but only execute in kernel mode.
- Older BSD-systems only had two *kernel processes* (swapper and pagedaemon) but freeBSD has many more.

Idle runs when there is nothing else to do.

swapper schedules the loading of processes into main memory.

pagedaemon executes the replacement algorithm for the virtual memory.

pagezero maintains a supply of zeroed pages.

Syncer ensures that dirty file data is written after 30 seconds.

The first user mode process is **init.** It is the first normal process that is started and is the origin of all other user mode processes.

System Entry

The kernel can be entered in three different ways:

- Hardware interrupt
- Hardware trap - exception
- Software generated trap - system call

All calls to the kernel use the interrupt system and are separated by use of different interrupt vectors.

There are three major kinds of handlers in the kernel:

1. Syscall(), for system calls.
2. Trap(), for exceptions and software-initiated traps other than system calls.
3. Device-driver interrupt handlers for hardware interrupts.

Run-time organization

- The kernel can be logically divided into *top half* and *bottom half*.
- *Top half* is called from system calls or traps.
- *Bottom half* is called via hardware interrupts.
- Activities in the *bottom half* are asynchronous with respect to the *top half* and can not depend on having a specific process running.
- The *top half* and *bottom half* of the kernel communicate through work queues.
- Data structures that are referenced from both *top half* and *bottom half* becomes critical regions that must be protected.
- In FreeBSD 5.2 the work queues are protected by a mutex.

System Calls

The system-call handler must do the following work:

- Verify that the parameters to the system call are located at a valid user address and copy them from the user's address space into the kernel.
- Call a kernel routine that implements the system call.

If a system call fails, the system call routine will set the C *errno* variable. At return from a system call *errno* is copied into a register. This register is copied into the caller's errno by the library routine that performs the system call.

A special error is that a system call is interrupted by a signal. In this case *errno* is set to EINTR.

Returning from a System Call

At return from a system call, several checks are made.

- Check if a signal have arrived. If a signal have arrived signal processing is performed.
- Check if any process has a higher priority than the running one. If this is the case, the context-switch routine is called to cause the higher priority process to run.
- If profiling have been requested, the time spent in the system call is calculated.

Software Interrupts

- Interrupt routines should be as short as possible to avoid blocking the interrupts for too long time.
- A way to reduce the execution time in the interrupt handler is to do the less time-critical processing at a lower priority.
- The mechanism for doing lower-priority processing is called a *software interrupt*.
- In FreeBSD 5.2 each software interrupt has a process context associated with it.
- The interrupt routine will create a queue of work to be done at software-interrupt level.
- The software interrupt processes are given a scheduling priority lower than the hardware interrupts, but higher than any user level process.
- An important use for software interrupts are the delivery of network packets to the destination process.

Clock Interrupts

- Clock interrupts are generated by a timer, typically every 10 milliseconds.
- Each interrupt is referred to as a *tick*.
- The clock interrupt usually is the highest priority interrupt in the system.
- At every clock interrupt the *hardclock()* routine is called.
- So the time spent in *hardclock()* is minimized, less time-critical processing is handled by a lower priority software-interrupt process called *softclock().*

Clock Interrupts Cont.

The work done by *hardclock()* is as follows:

- Increment the current time of day.
- If the currently running process has a virtual or profiling interval timer, increment the timer and deliver the signal if the timer has expired.
- If the system does not have a separate clock for process profiling, the *hardclock()* routine does the operations normally done by *profclock()*.
- If the system does not have a separate clock for statistics gathering , the *hardclock()* routine does the operations normally done by *statclock()*.
- If *softclock()* needs to be run, make the softclock process runnable.

Statistics and Process Scheduling

- Resource utilization statistics may be used to determine future scheduling priorities and can also be used to measure the execution time for different routines in a program (profiling).
- Using the *hardclock()* to collect statistics can give incorrect data, because processes can become synchronized with the system clock.
- On the PC, a statistics clock is run at a different frequency than the hardclock.
- The FreeBSD *statclock()* runs at 128 ticks per second.

Statclock() does the following:

- Add a tick to the executing process. If a process has accumulated four ticks, recalculate its priority and possibly arrange for a context switch.
- Collect statistics on what the system was doing at the time of the tick.

Timeouts

- *The Softclock()* routine processes timeout requests.
- The data structure that describes waiting events is called the *callout queue* (fig. 3.2).
- At timeout, a specified subroutine is called.
- Insertion into the *callout queue* is done with the following subroutine:

```
timeout(ftn, arg, to_ticks)
   timeout_t *ftn; /* call ftn at timeout */
   void *arg;
   int to_ticks; /* number of  ticks till timeout */
```

Softclock() and the *callout queue* can be used for the following purposes:

- Recalculating the process priorities (Done once per second)
- Retransmission of network packets.
- Watchdog timer for peripherals that require monitoring.
- Process real-time timer (Section 3.6).

Organization of the callout queue

- Older BSD systems used one queue sorted in time order.
- Insertion in this queue was O(n) and removal was O(1).
- FreeBSD 5.2 uses a method that gives O(1) in the normal case for both insertion and removal.
- The queue consists of a table with 200 list heads.
- A pointer labeled *now* points to the list head that represents current time.
- At insertion into the queue, the absolute time for timeout is stored in the callout struct.
- The next list head in the table represents the time now+1 and so on up to now+199.
- Time is measured in *ticks* and current time is represented by the global variable *tick*, which is updated by the *hardclock()*.

Organization of the callout queue cont.

- The *now* pointer is incremented by *hardclock()* at every tick. If the list pointed to by *now* is nonempty, the *softclock()* process is scheduled to run.
- *Softclock()* compares the point of time stored in the callout struct with current time. If these points of time matches the subroutine stored in the callout struct is called.
- When an event n ticks in the future is inserted, it is placed in the list with index (now+n) mod 200.

Timing Services for Processes

The kernel provides the following timing services to processes via system calls:

Gettimeofday: Returns the time of day given in the number of microseconds from epoch (January 1, 1970). On most processors including the PC, the time value is derived from a battery-backup time-of-day register.

Settimeofday: Sets the time-of-day time. May result in time running backwards.

Adjusttime: Adjusts the time-of-day time without time running backwards.

Setitimer/getitimer: Set or read an interval timer. Three timers exist giving *real*, *virtual* or *profile* time. The *real* timer measures real time, *virtual* measures the process's execution time in user mode and *profile* measures execution time in both user and kernel mode. When a timer expires, a signal is sent to the process.

Identification of users

- Users are identified by a 32-bit number called *user identifier* (UID).
- A user may also belong to one or more groups.
- A group is identified by a 32-bit group identifier (GID).
- Each file has three sets of permission bits for each owner group and other.
- There is also the *setuid* bit.
- UID and GID are inherited from the parent process.
- At login UID and GID are set by the login program.

Handling of UID in FreeBSD

- In FreeBSD UID is stored in three places, called *real UID*, *effective UID* and *saved UID*.
- Real UID is the original UID of the process.
- *Effective UID* is used for checking permissions and is set to *real UID* for normal programs and to the UID of the owner of the program file for setuid programs.
- At exec, *saved UID* are set to the same value as *effective UID*.
- The system call *seteuid()* assigns *effective UID* from either *real UID* or *saved UID* (see table 3.2). Can be used by setuid programs to regain normal privileges when the extra privileges are not needed any more.

Sessions and Process groups

- Every process belongs to a *process group.*
- Certain signals (for example SIGINT) are sent to all processes in the same *process group*.
- A *session* is a collection of *process groups*.
- Every *session* has a *controlling process* (normally a login shell) and is associated with a terminal, known as its *controlling terminal*.
- At any point of time, the *controlling terminal* is connected to exactly one *process group*.
- When the *controlling process* exits, access to the terminal is taken away from any remaining processes within the *session*.

Resource Utilization

The resources used by a process are returned by the system call *getrusage()*.

Examples of such system resources are:

- The amount of user and system time used by the process.
- The memory utilization of the process.
- The paging and disc I/O activity of the process.
- The number of voluntary and involuntary context switches taken by the process.
- The amount of interprocess communication used by the process.