Processes

- Processes execute in *user mode* (application code) or *kernel mode* (operating system code).
- The *user mode* state consists of the processor registers and of the content of the memory segments.
- The *kernel mode* state consists of the processor registers and the information the kernel stores about the process.
- The kernel state for each process is divided into several different data structures (fig. 4.1).
- The top level data structures are the *process structure* and the *user structure*.
- In old Unix systems the *user structure* contained data that was not needed when the process was swapped out.
- In FreeBSD the *user structure* is used for only a couple of structures referenced from the *process structure*.
- A new *thread structure* has been introduced in FreeBSD to keep information about a single thread.

Process structure

The *process structure* contains:

- Process identification (PID)
- signal state
- timers: real-time and CPU-utilization counters
- Pointers to the substructures

Process structure, cont.

The process substructures (fig. 4.1) have the following categories of information:

- Process-group identification
- User credentials (UID, GID)
- Memory management
- File descriptors
- Resource accounting
- Statistics
- Signal actions
- Thread structure

Processes can be in any of the following states: NEW, NORMAL, or ZOMBIE.

A process in NORMAL state can be RUNNABLE, SLEEPING or STOPPED.

Thread structure

The *thread structure* contains the following categories of information:

- Scheduling: for example thread priority
- Thread state: (runnable, sleeping), and *wait channel* if sleeping.
- Machine state: the machine dependent thread information
- TCB (Thread Control Block): the user and kernel mode execution states including MMU registers
- Kernel stack: the per-thread execution stack for the kernel

Process Lists

- The kernel keeps track of a process by inserting its process entry (object of type process struct) into different lists.
- Process entries are on the *zombproc* list if the process is in the ZOMBIE state.
- Other processes are on the *allproc* list
- Most threads, except the currently running thread are also in one of two queues: the *run queue* or the *sleep queue*.
- The run queues are organized according to the scheduling priority.
- The sleep queues are organized in data structures that are hashed by event identifier (wait channel).

Scheduling - priorities

- CPU time is made available to the processes based on their scheduling class and scheduling priority.
- The FreeBSD kernel has two kernel and three user mode scheduling classes (fig. 4.2).
- A thread has two scheduling priorities: one for scheduling user-mode execution (kg_usr_pri) and one for scheduling kernel mode execution (td_priority).
- Priorities range between 0 and 255, with lower values interpreted as a higher priority.
- User mode priorities range from 128 to 255.
- Priorities less than 128 is only used when a thread is *asleep* in the kernel and immediately after the thread is awakened.
- A thread that is *asleep* will be awakened by a signal only if it sets the PCATCH flag when *msleep()* is called.

User structure

Historically, the *user struct* has been located at a fixed virtual address in the kernel, that was remapped at every context switch. There were three reasons for this:

1. In many architectures *user struct* could be mapped on top of the process address space. This simplified saving of process state.
2. *User struct* could always be addressed on a fixed address via u.xxx.
3. When a process forks, its kernel stack is copied. Because the kernel stack was located in the user struct, it was located at the same virtual address in the child and parent processes.

On architectures with virtual address caches, mapping *user struct* to a fixed address is slow and inconvenient. For this reason *user struct* is not located at a fixed virtual address in FreeBSD 5.2.

A problem is that the kernel stack, now located in the *thread struct*, will get different virtual addresses in the child and parent processes. In FreeBSD 5.2 this problem is solved by eliminating all but the top call frame from the child's stack after it is copied from the parent so that it returns directly to user mode.

Context Switching

- A *voluntary* context switch occurs when a thread blocks because it requires a resource that is unavailable.
- An *involuntary* context switch takes place when a thread executes for the duration of its time slice or when the system identifies a higher-priority thread.
- Voluntary context switches are synchronous with respect to the currently executing thread whereas involuntary context switches are asynchronous.
- Voluntary context switches are initiated by calling the kernel subroutine *msleep()*.
- An involuntary context switch is forced by direct invocation of the low-level context-switching routines *mi_switch()* and *setrunnable()*.

Thread State

Context switching requires that *user-mode* and *kernel-mode* state is changed.

To simplify context switching, the complete *user-mode* state is stored in *thread struct* when the process is not executing in user mode.

Process state:

- **Kernel-mode hardware-execution state.** Saved in *thread struct TCB* at context switches.
- **User-mode hardware-execution state.** Saved at kernel stack (thread struct) every time the kernel is entered.
- **process struct.** Always memory resident.
- **Memory resources.** Described by the memory-management registers which are saved in thread struct TCB at every context switch. As long as the process remains in memory, these values will remain valid.

Low level context switches are performed by the dispatching routine *mi_switch()*.

Voluntary Context Switching

- When *msleep()* is called a *priority* and a *wait channel* are given as parameters.
- The priority specified in msleep() is the priority that should be assigned to the process when it is awakened and is only used in kernel mode.
- The *wait channel* is typically the address of some data structure that identifies the resource or the event the process is waiting for.
- When an event occurs, all threads sleeping on that *wait channel* will be awakened by a call to *wakeup()*.

Voluntary Context Switching cont.

In some cases a special wait channel is used.

- The global variable *lbolt* is awakened once per second.
- When a parent process does a wait system call, it will sleep on its own process structure. When a child terminates it will wake the parent's process structure.

Msleep() - implementation

Sleeping processes are organized in an array of queues (fig. 4.3).

Msleep() and wakeup() hash the wait channel to calculate an index into the sleep queues.

Msleep() takes the following steps in its operation:

1. Block interrupts by acquiring the s*ched_lock* mutex.
2. Save *wait channel* in thread struct and hash the *wait channel* to locate a sleep queue for the thread.
3. Set the thread's priority to the value it will have when it is awakened and set the SLEEPING flag.
4. Put the thread at the end of the sleep queue selected in 2.
5. Call *mi_switch()* to request that another thread is scheduled. The *Sched_lock* mutex is released as part of switching to the other thread.

Wakeup() - implementation

Wakeup() processes the entire sleep queue for the specified *wait channel*. For every thread that need to be awakened it does the following:

1. Removes the thread from the sleep queue.
2. Recompute the user-mode priority if the thread has been sleeping for more than one second.
3. If the thread is in SLEEP state, place it in the run queue (for other cases, see the book).

If a process that *wakeup()* moved to the run queue has higher priority than the currently executing process, *wakeup()* will also request that the CPU is rescheduled as soon as possible.

Synchronization

- Older BSD kernels only supported one processor.
- In this case locking was needed only due to interrupts and if *msleep()* was called.
- The result was that a simple locking algorithm could be used.
- Symmetric Multiprocessors (SMP) are much more demanding on locking.
- In the first BSD kernels with support for SMP, a "*giant lock*" was used to lock the whole kernel.
- This mechanism do not allow any parallelism at all and gives unacceptable performance on a SMP.
- In FreeBSD 5.0 more fine grained locking was introduced.

Synchronization in old single-processor Kernels

The access to internal data structures was synchronized by two flags, LOCKED and WANTED.

Protocol to request access to a resource (data structure):

```
if LOCKED
    set WANTED;
    sleep(WCHAN_X, priority);
else
    set LOCKED;
        use resource;
    clear LOCKED;
    if WANTED wakeup(WCHAN_X);
end if;
```

For access to data structures that were called from interrupt level (bottom half), the priority level also needed to be raised.

```
s = splbio(); /* raise priority */
     use buffer;
splx(s);      /* restore priority */
```

Mutex Synchronization

- The FreeBSD 5 locking mechanism is modeled after Posix threads and use *mutexes* and *condition variables*.
- There are two variants of mutexes, spin mutexes (busy wait) and sleep mutexes (context switch).
- In principle, a spin mutex is preferred if the waiting time for the lock is shorter than two context switches.
- The strategy in FreeBSD 5 is to use sleep mutexes as default.
- Implementation of the mutex operations requires atomic instructions such as *test_and_set*.
- The interrupts are always disabled when a process is holding a spin lock.
- It is not allowed to go to sleep while holding a spin mutex
- A mutex is owned by the process that locked it and can only be unlocked by this process.

Mutex procedures

All the procedures take a lock variable as parameter.

The most common operations on mutexes are:

mtx_init() Must be called to initiate the *mutex* before it can be used.

mtx_lock() Tries to lock the *mutex*. It the *mutex* is already locked, a context switch is done.

mtx_lock_spin() Similar to *mtx_lock*, but spins instead of doing a context switch.

mtx_trylock() Tries to lock a *mutex*. Returns 1 if the locking succeeded and otherwise 0.

mtx_unlock() Unlock a *mutex*. If a process with higher priority is waiting on the *mutex*, a context switch is done.

mtx_unlock_spin() Unlocks a spin *mutex*. The interrupt state is restored to its state before the lock was acquired.

Lock-manager locks

- For situations where reading is much more common than writing, so called shared/exclusive locks are suitable.
- This type of locks are also called reader/writer locks.
- In FreeBSD there is a *lockmgr* routine to handle such locks.
- Lockmgr allows the following operations:
  - → shared
  - → exclusive
  - → upgrade
  - → exclusive upgrade
  - → downgrade
  - → release

Synchronization and deadlock

- Because FreeBSD now supports SMP, there is always the risk of deadlock, if a process i holding more than one lock.
- To prevent deadlock, all resources are grouped into classes.
- Resources in the same class are selected so that it is always enough to lock one resource in the class.

Locking rules:

1. A thread may only acquire one lock in each class.
2. A thread may acquire a lock in a class with a higher number than the highest numbered class it already holds a lock in.

There is a *witness module* that checks that the locking order is followed.

Scheduling

- There are two different schedulers in FreeBSD.
- Which one to use is selected when the kernel is compiled.
- The new ULE scheduler is designed to get good SMP performance.
- The old scheduler uses a variant of multilevel feedback queues.
- The time quanta is 0.1 seconds.
- The priority of the processes is dynamically adjusted so that processes that use much CPU time gets their priority decreased and processes that are waiting get increased priority.

Priorities

The priority calculation utilizes two variables in *process structure*.

kg_estcpu estimates the process's CPU utilization.

kg_nice value between -20 and +20 that can be set by the user to increase (if superuser) or decrease the process's priority.

- User mode priority is stored in *kg_usr_pri* and is calculated every four clock ticks.
  - → $kg\_usr\_pri = PRI\_MIN\_TIMESHARE + kg\_estcpu/4 + 2 * kg\_nice \qquad (Eq.\ 4.1)$
- The *hardclock()* routine, which is called every10 ms increments *kg_estcpu* for the executing process.
- When *kg_estcpu* for a process has been incremented four times, the *setpriority()* routine is called to recalculate the priority according to equation 4.1.

Priorities, cont.

In addition, *kg_estcpu* for runnable processes is adjusted once a second via a digital decay filer run by the *schedcpu()* routine:

$$kg\_estcpu = \frac{2 * load}{2 * load + 1} kg\_estcpu + kg\_nice \quad (4.2)$$

- *load* is the average number of processes in the ready queue during the previous minute.
- For processes that have been sleeping for more than 1 second, the adjustment of *kg_estcpu* is calculated by *wakeup()* when the process is awakened using the formula:
- $kg\_estcpu = \left[\frac{2*load}{2*load+1}\right]^{kg\_slptime} * kg\_estcpu \, (Eq.\ 4.3)$
- *Kg_slptime* is the number of seconds the process have been blocked.
- The effect of the decay filter is that 90% of the CPU utilization is forgotten after 5 seconds.

Run Queues and Context Switching

- The scheduler uses 64 queues that are selected in priority order (fig. 4.6).
- As there are 256 different priorities, the correct run queue is selected by dividing the priority with 4.
- To save time, the threads on a specific queue are not sorted in priority order.
- A 64 bit bit-vector, *rq_status*, is used to identify nonempty queues.
- Context-switching is done by the routine mi_switch() that calls the machine-dependent optimized routine cpu_switch() to do the actual work.
- If the context-switch is initiated from top-half *mi_switch()* can be called immediately from *msleep()*.
- If the context-switch is due to an interrupt, the interrupt thread will usually not want to call *mi_switch()* itself but instead it sets a NEEDRESCHED flag and activates a software interrupt that will call *mi_switch()* at the conclusion of the interrupt.
- In architectures that do not support software interrupts, a reschedule flag is tested at the return from every system call, trap and interrupt.

The ULE Scheduler

Goals:

- Support for SMP (Symmetric Multi Processing)
- Give O(1) scheduling time. That is, the scheduling time do not depend on the number of threads in the system.
- Support processor affinity in SMP systems.

Processor affinity means that a process is not moved to another processor unless there is a very strong reason to do so.

ULE - Background

- To move a process from one processor to another is expensive because all data that the process had in caches need to be reloaded.
- The problem is somewhat smaller with SMT (Symmetric Multi Threading), also called hyperthreading.
- Here all the threads share memory and caches.

ULE - Implementation

- The ULE scheduler do not use priorities to get a fair division of processing time between the processes, instead two queues are used that are switched when the current queue becomes empty.
- A process is placed in a certain queue until it blocks or its time quanta runs out.
- For every CPU there are three queues.

Idle is only used for the *idle-thread* that executes only when the other two queues are empty.

current Interactive threads, interrupt- and realtime-threads are placed here. Processes are scheduled in priority order from this queue until it becomes empty.

next Noninteractive threads are placed here. When the current queue becomes empty, *next* and *current* are switched.

ULE - Implementation Cont.

- A thread is considered to be interactive if the ratio between its voluntary sleep time versus its run time is below a certain threshold.

m = (maximum interactivity_score)/2

Interactivity_score is calculated by the formulas:

sleep>run interactivity_score = m / (sleep/run)

sleep<run interactivity_score = 2m - m / (run/sleep)

- Threads with an interactivity_score below an experimentally chosen threshold are considered to be interactive.
- The interactivity_score is calculated by the subroutine *sched_interact_update()*, which is called at several occasions - for example when a thread is awakened by a *wakeup()* call.

ULE - Load Balancing

- ULE uses two methods to balance the load between the CPU:s.
- Whenever a processor is idle it sets a bit in a global bitmask.
- Whenever an active CPU is about to add work to its own run queue, it first checks to see if it has excess work and if another processor is idle.
- If an idle processor is found, the process is migrated to the idle processor using an IPI (Inter Processor Interrupt).
- The other method is "*push migration*" that is performed two times per second by *sched_balance()*.
- *Sched_balance()* picks the most-loaded and the least-loaded processors in the system and equalizes their run queues.

Process Creation

- New processes are created with fork, vfork or rfork.
- The *fork* system call creates a complete copy of the parent.
- The *rfork* system call creates a new process that shares a selected set of resources with its parent.
- The *vfork* system call shares the page tables for the code and data segments with the parent.

All fork system calls involves three main steps:

- Allocating and initializing a new *process structure* for the child process.
- Duplicating the context of the the parent for the child process (including thread struct and virtual memory resources).
- Place the child on the correct run queue.

Process Termination

Processes terminate through an *exit* system call or as the result of a signal.

Within the kernel, a process is terminated by calling the *exit()* subroutine:

First all other threads that belong to the process are terminated in the following way:

- Any thread entering the kernel from user space will invoke *thread_exit()* when it traps into the kernel.
- Threads already in the kernel calling *sleep* will return immediately with EINTR or EAGAIN. When they try to return from the kernel *thread_exit()* is called.

After this *exit()* cleans up the kernel mode execution state:

- Cancels any pending timers.
- Releases virtual-memory resources.
- Closes open descriptors.
- Handles stopped or traced child processes.

Process Termination cont.

The process is moved from the *allproc* list to the *zombie* list.

The *exit()* routine when does the following:

- Records the termination status.
- Bundles up a copy of the process's resource usage.
- Notifies the parent process.

Normally, the parent has called *wait4* and is awakened.

The wait4 call will search for child processes in ZOMBIE state. For every ZOMBIE process that matches the wait criterion, the termination status is copied and the process entry is freed.

Signals

- FreeBSD signals are designed to be software equivalents of hardware interrupts or traps.
- Each signal has an associated default action that specifies how it should be handled if nothing else is requested (Tab 4.4).
- A program can request an alternate handling by using the *sigaction* system call:
  - → Ignore the signal
  - → Call a user-written *signal handler*
  - → Taking the default action
- Like hardware interrupts, signals can be temporarily blocked by using the system call *sigprocmask*.
- If the process have several threads, the handling of signals is independently specified by each thread.
- The system call *sigalstack* can be used to specify that a special signal stack shall be used for signal handling.

Implementation of signals

- The implementation of signals is broken up in two parts: posting a signal to a process and delivering it to the target thread.
- A signal can be posted at any time from any code in the kernel by calling the *psignal()* subroutine.
- A posted signal is normally added to the set of pending signals for the appropriate thread.
- When a signal is raised because of the action of the currently executing thread, it is only delivered to that thread.
- Other signals are delivered to the first thread that do not have it masked.
- Due to many special cases, the signal handling is complicated (See the book).
- Delivery of signals is normally possible only when the thread is executing.

Delivery of signals

- Each time a thread returns from a call to sleep() (with the PCATCH flag set) or from a system call, it is checked if a signal is pending delivery.
- Delivery of signals are done by the following code to be found in many places in the kernel code:

```
if (sig = CURSIG(curthread))
    postsig(sig);
```

- CURSIG is a macro that examines if there is an unmasked signal in *td_siglist* for the running thread. If this is the case, the number of the signal is returned.
- If delivery of the signal results in the process being terminated, this is done by CURSIG otherwise *postsig()* is called.
- Postsig() takes care of two special cases:
  - → Producing a core dump
  - → Invoking a signal handler (Fig. 4.8)
- *Postsig()* is implemented by calling the machine dependent *sendsig()* routine.

Process Groups and Sessions

- A *process group* is a collection of related processes, such as a shell pipeline, that have the same *process-group identifier*.
- A process is always a member of a single process group.
- When a process is created, it is placed in the same *process group* as its parent.
- A process can change its own process group or that of a child process using the *setpgid* system call.
- If a process calls *setpgid* to set its process-group identifier to the same value as its PID, a new *process group* is created with the process as *process-group leader*.
- When a shell creates a new process to execute a command, the process calls *setpgid* to set its process-group identifier to the same value as its PID before executing the command.
- This creates a new *process group* with the process as *process-group leader*.
- If the command is part of a pipeline, each additional process in the pipeline will call *setpgid* to join the existing process group.
- For each new process group, the kernel allocates a *process-group structure (p_pgrp)*.

Sessions

- A *session* is a collection of one or more process groups.
- Normally, all processes created by the same login shell belongs to the same *session*.
- A process ,usually a login shell, may create a new session by calling the *setsid* system call, becoming the *session leader* for the session.
- A session may have an associated *controlling terminal* that is used for communicating with the user.
- Only a session leader may allocate a controlling terminal, becoming a *controlling process* when it does so.
- The controlling terminal communicates with the process group executing in the *foreground*.
- All other process groups in the session executes in the *background*.

Job Control

- The original reason that the complex mechanisms with process-groups and sessions were added to the system, was the job control mechanism originally implemented for the C shell in BSD4.1 Unix.
- The process groups are also important for the signal handling and for window systems like X.
- The *Job control* mechanism allows a *job* (that is a process group) to be stopped (with CTRL Z) and restarted (with fg).
- A stopped job can also be started in the background with the command *bg*.
- Signals that are generated from the key board, are sent to all processes in the *foreground* process group.

Job Control cont.

- If a controlling process exits (the user logs out), the system revokes further access to the the controlling terminal and sends a SIGHUP signal to the foreground process group.
- When a job-control shell exits, each process group that it created becomes an *orphaned process group*.
- Each such orphaned process group is sent a SIGHUP signal and a CONTINUE signal if any of its members are stopped.
- Background processes that catch or ignore hangup signals can continue to execute after the controlling process has terminated.