FreeBSD Internals
Day 2

Joel Isaacson
Ascender Technologies Ltd

Copyright 2007
This work is licensed under the

Creative Commons Attribution License.

Process Layout

- Each process starts with four memory areas: text, bss, data and stack.
- Other memory areas are added using the mmap system call.
- The shared library facility is implemented via mmap by a method that is very similar to the method used by linux.

Process Layout

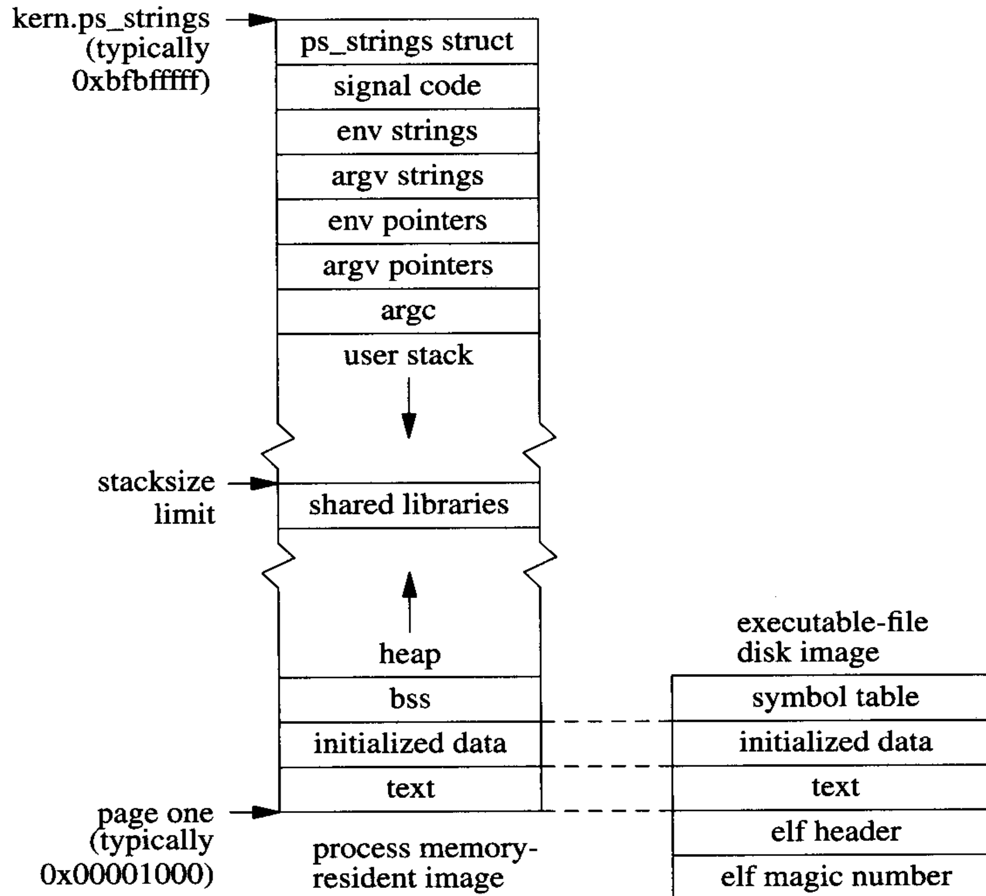
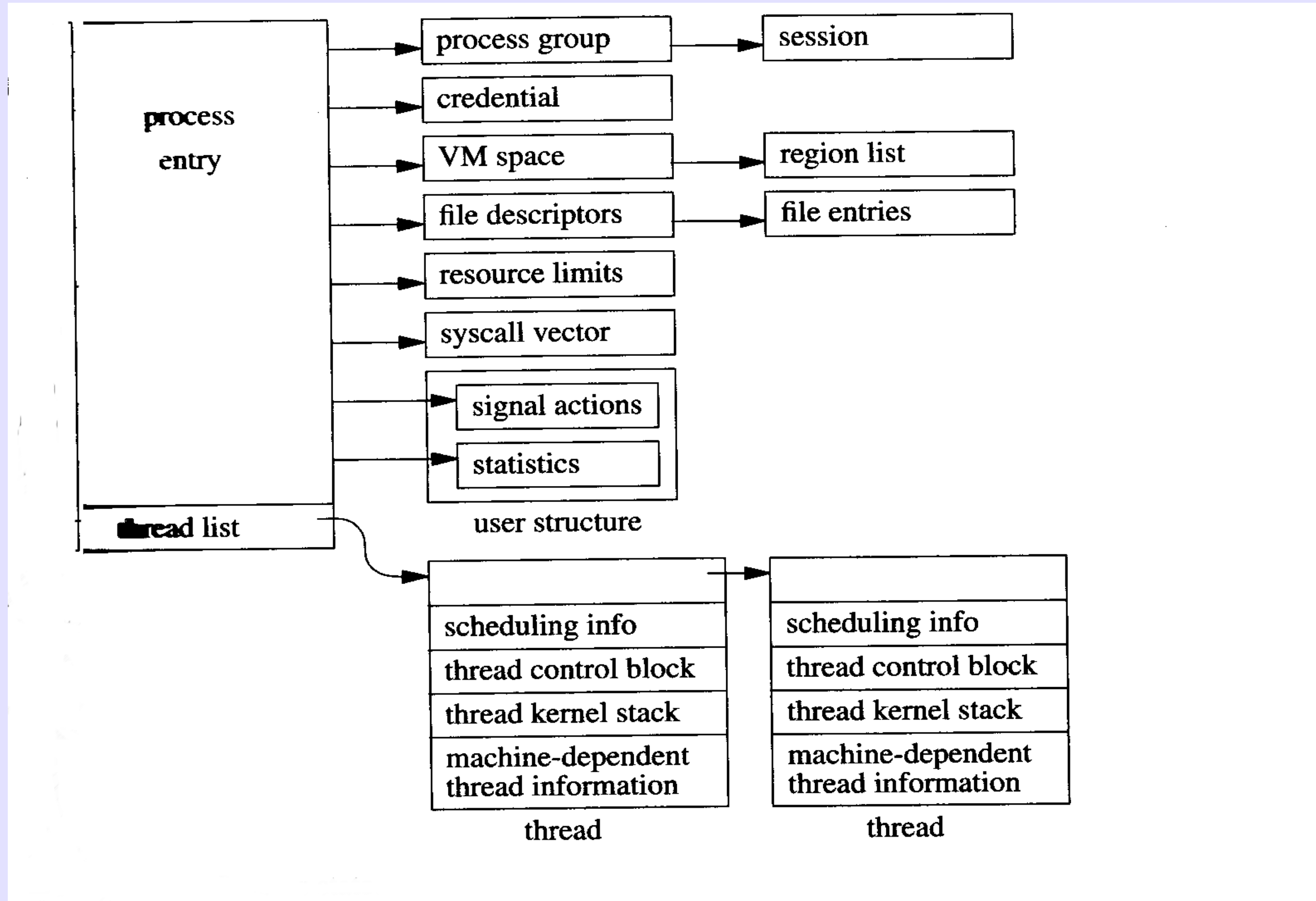


Figure 3.3 Layout of a FreeBSD process in memory and on disk.

Execution Framework

- Executable files have their execute bits set.
- Each executable file has an exec header with magic numbers for a particular type of executable.
- There are two types of execution models: direct or interpreter.

Process State



Thread Scheduling Classes

: Thread-scheduling classes.

Range	Class	Thread type
0 – 63	ITHD	Bottom-half kernel (interrupt)
64 – 127	KERN	Top-half kernel
128 – 159	REALTIME	Real-time user
160 – 223	TIMESHARE	Time-sharing user
224 – 255	IDLE	Idle user

Context Switching

- The kernel switches between threads in an effort to share the CPU efficiently.
- A thread can be preempted from running by:
  - Voluntary – the thread gives up the CPU
  - Involuntary – the thread is preempted without it's cooperation.

Voluntary Context Switch

- Context switching takes place only in kernel mode.
- This context switch is initiated by the *sleep()* routine.
- This context switch is synchronous.

Voluntary Context Switch

- This occurs when a thread must await the availability of a resource or an event.
- For example a thread will block when it wants to read from a device and the data is currently not available.
- The thread is then put into the sleep queue.

tsleep()

int tsleep(void *ident, int priority, const char *wmesg, int timo);

The parameter ident is an arbitrary address that uniquely identifies the event on which the process is being asleep. All processes sleeping on a single ident are woken up later by wakeup().

msleep()

```
int msleep(void *ident, struct mtx *mtx, int priority, const char *wmesg,int timo);
```

The msleep() function is a variation on tsleep. The parameter mtx is a mutex which will be released before sleeping and reacquired before msleep() returns. The mutex is used to ensure that a condition can be checked atomically,

wakeup()

```
void wakeup(void *ident);
```

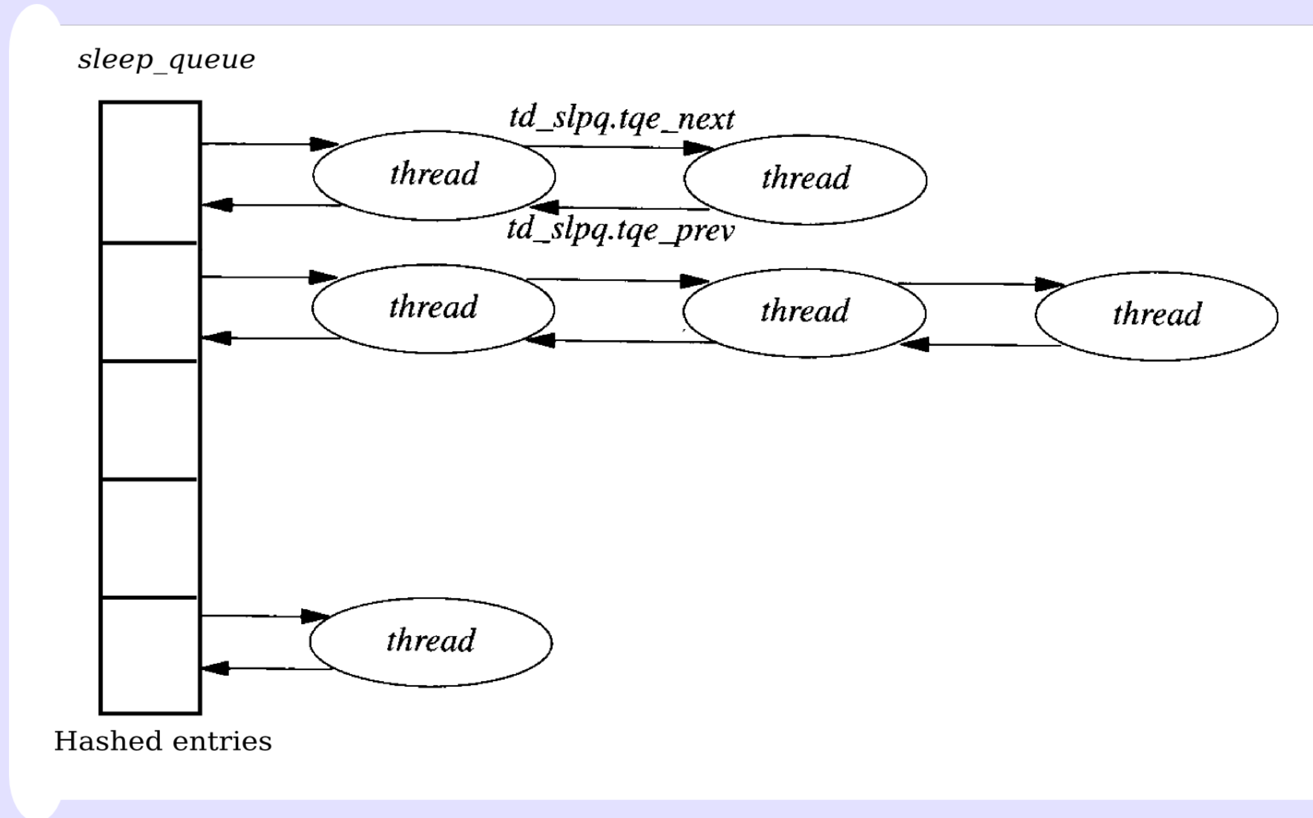
All processes sleeping on a single ident are woken up later by wakeup(), often called from inside an interrupt routine, to indicate that the resource the process was blocking on is available now.

wakeup_one()

```
void wakeup(void *ident);
```

The wakeup_one() function is used to make the first process in the queue that is sleeping on the parameter ident runnable. This can prevent the system from becoming saturated when a large number of processes are sleeping on the same address.

sleep_queue



Special sleep addresses

- There are special addresses for sleep:
  - The variable lbolt is “woken-up” every second.
  - When a process exits a wakeup is done on it's parent's process structure. This is used to implement *wait4()*.
  - When a process get a signal the p_sigacts element is woken up. This is used to implement *sigsuspend()*.

Involuntary Context Switch

- This context switch can be either synchronous (traps) or asynchronous (interrupts).
- The actual context switch is performed by the mi_switch() routine.
- This causes the highest priority thread to be run.

Synchronization

- Freebsd provides many forms of synchronization algorithms.
- Synchronization is used to guarantee that shared data structures are consistently accessed
- SMP need significantly different synchronization primatives.

Synchronization

Level	Type	Sleep	Description
Lowest	hardware	no	memory-interlocked test-and-set
	spin mutex	no	spin lock
	sleep mutex	no	spin for a while, then sleep
	lock manager	yes	sleep lock
Highest	witness	yes	partially ordered sleep locks

Mutex's

Mutexes are the most basic and primary method of thread synchronization. The mtx_init() function must be used to initialize a mutex before it can be passed to any of the other mutex functions.

Mutex's

void mtx_init(struct mtx *mutex, const char *name, const char *type, int opts);

The name option is used to identify the lock in debugging output etc. The type option is used by the witness code to classify a mutex when doing checks of lock ordering.

Mutex's

```
void mtx_init(struct mtx *mutex, const char
*name, const char *type, int opts);
```

The data pointed to must remain stable until the mutex is destroyed. The opts argument is used to set the type of mutex. It may contain either MTX_DEF or MTX_SPIN but not both.

Mutex's

void mtx_lock(struct mtx *mutex);

The mtx_lock() function acquires a MTX_DEF mutual exclusion lock on behalf of the currently running kernel thread. If another kernel thread is holding the mutex, the caller will be disconnected from the CPU until the mutex is available (i.e., it will block).

Mutex's

void mtx_lock_spin(struct mtx *mutex);

The mtx_lock_spin() function acquires a MTX_SPIN mutual exclusion lock on behalf of the currently running kernel thread. If another kernel thread is holding the mutex, the caller will spin until the mutex becomes available. Interrupts are disabled during the spin and remain disabled following the acquiring of the lock.

Mutex's

```
int mtx_trylock(struct mtx *mutex);
```

The mtx_trylock() attempts to acquire the MTX_DEF mutex pointed to by mutex. If the mutex cannot be immediately acquired mtx_trylock() will return 0, otherwise the mutex will be acquired and a non-zero value will be returned.

Mutex's

void mtx_unlock(struct mtx *mutex);

The mtx_unlock() function releases a MTX_DEF mutual exclusion lock. The current thread may be preempted if a higher priority thread is waiting for the mutex.

Mutex's

void mtx_unlock_spin(struct mtx *mutex);

The mtx_unlock_spin() function releases a MTX_SPIN mutual exclusion lock. The mtx_trylock() attempts to acquire the MTX_DEF mutex pointed to by mutex. If the mutex cannot be immediately acquired mtx_trylock() will return 0, otherwise the mutex will be acquired and a non-zero value will be returned.

The Lock Manager

- The *lockmgr()* function handles general locking functionality within the kernel, including support for shared and exclusive locks, and recursion.
- *lockmgr()* is also able to upgrade and downgrade locks.
- It is used for interprocess synchronization.

SX

- Shared/exclusive locks are used to protect data that are read far more often than they are written.
- Mutexes are inherently more efficient than shared/exclusive locks, so shared/exclusive locks should be used prudently.

Condition Variables

- Condition variables are used in conjunction with mutexes to wait for conditions to occur.
- When a thread waits on a condition, the mutex is atomically released before the thread is blocked, then atomically reacquired before the function call returns.