CHAPTER 9

Embedded Operating Systems

In This Chapter

- *Define operating system*
- *Discuss process management, scheduling, and inter-task communication*
- *Introduce memory management at the OS level*
- *Discuss I/O management in operating systems*

An operating system (OS) is an optional part of an embedded device's system software stack, meaning that not all embedded systems have one. OSes can be used on any processor (ISA) to which the OS has been ported. As shown in Figure 9-1, an OS either sits over the hardware, over the device driver layer or over a BSP (Board Support Package, which will be discussed in Section 9.4 of this chapter).

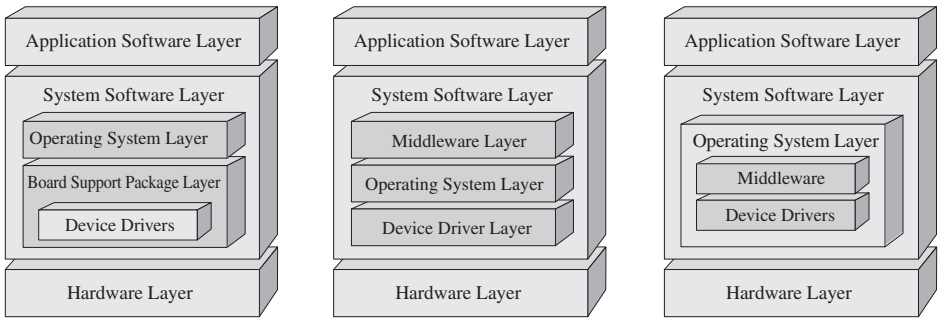


Figure 9-1: OSes and the Embedded Systems Model

The OS is a set of software libraries that serves two main purposes in an embedded system: providing an abstraction layer for software on top of the OS to be less dependent on hardware, making the development of middleware and applications that sit on top of the OS easier, and managing the various system hardware and software resources to ensure the entire system operates efficiently and reliably. While embedded OSes vary in what components they

possess, all OSes have a ***kernel*** at the very least. The kernel is a component that contains the main functionality of the OS, specifically all or some combination of features and their interdependencies, shown in Figures 9-2a-e, including:

- **Process Management**. How the OS manages and views other software in the embedded system (via *processes*—more in Section 9.1, Process Management). A subfunction typically found within process management is *interrupt and error detection management*. The multiple interrupts and/or traps generated by the various processes need to be managed efficiently so that they are handled correctly and the processes that triggered them are properly tracked.
- **Memory Management**. The embedded system's memory space is shared by all the different processes, so access and allocation of portions of the memory space need to be managed (more in Section 9.2, Memory Management). Within memory management, other subfunctions such as *security system management* allow for portions of the embedded system sensitive to disruptions that can result in the disabling of the system, to remain secure from unfriendly, or badly written, higher-layer software.
- **I/O System Management**. I/O devices also need to be shared among the various processes and so, just as with memory, access and allocation of an I/O device need to be managed (more in Section 9.3, I/O System Management). Through I/O system management, *file system management* can also be provided as a method of storing and managing data in the forms of files.

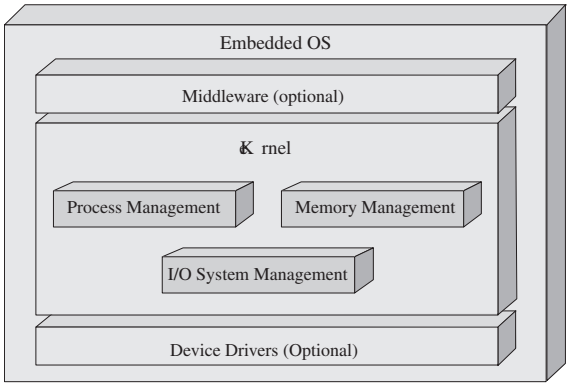


Figure 9-2a: General OS model

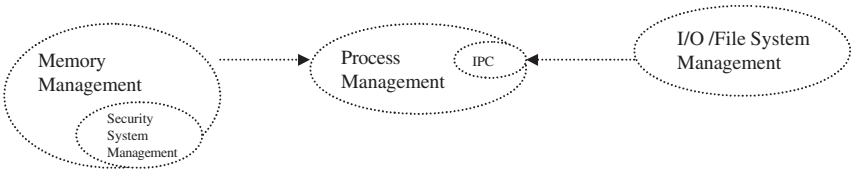


Figure 9-2b: Kernel subsystem dependencies

Because of the way in which an operating system manages the software in a system, using processes, the process management component is the most central subsystem in an OS. All other OS subsystems depend on the process management unit.

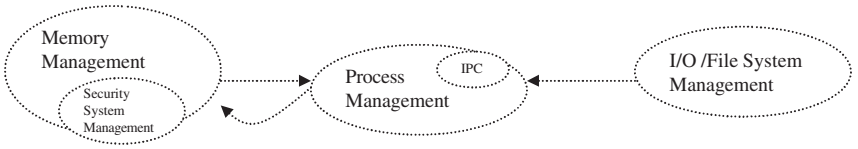


Figure 9-2c: Kernel subsystem dependencies

Since all code must be loaded into main memory (RAM or cache) for the master CPU to execute, with boot code and data located in non-volatile memory (ROM, Flash, etc.), the process management subsystem is equally dependent on the memory management subsystem.

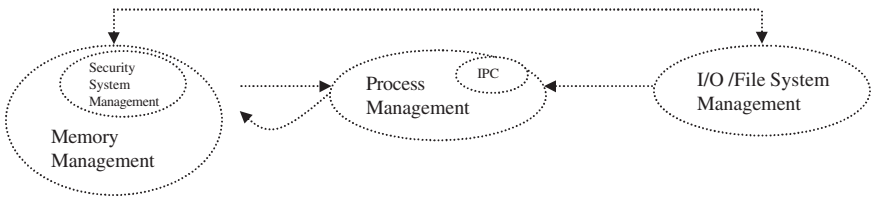


Figure 9-2d: Kernel subsystem dependencies

I/O management, for example, could include networking I/O to interface with the memory manager in the case of a network file system (NFS).

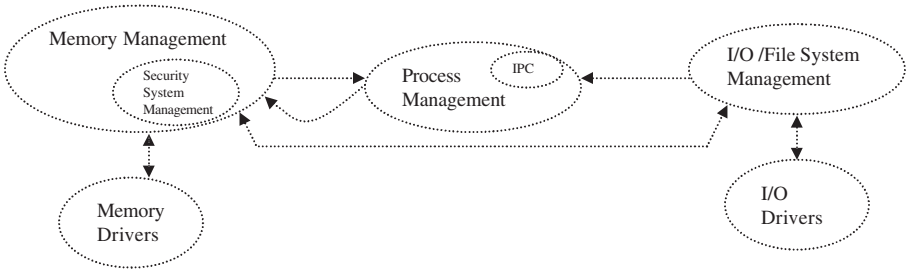


Figure 9-2e: Kernel subsystem dependencies

Outside the kernel, the Memory Management and I/O Management subsystems then rely on the device drivers, and vice-versa, to access the hardware.

Whether inside or outside an OS kernel, OSes also vary in what other system software components, such as device drivers and middleware, they incorporate (if any). In fact, most embedded OSes are typically based upon one of three models, the ***monolithic***, ***layered***, or ***microkernel*** (*client-server*) design. In general, these models differ according to the internal

design of the OS's kernel, as well as what other system software has been incorporated into the OS. In a *monolithic* OS, middleware and device driver functionality is typically integrated into the OS along with the kernel. This type of OS is a single executable file containing all of these components (see Figure 9-3).

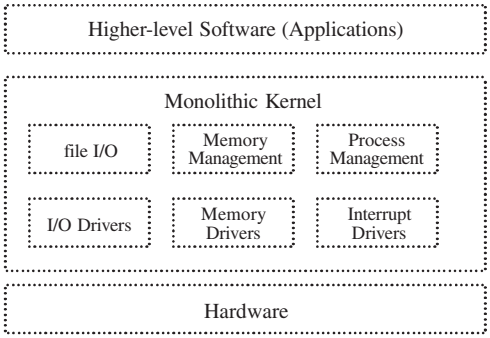


Figure 9-3: Monolithic OS block diagram

Monolithic OSes are usually more difficult to scale down, modify, or debug than their other OS architecture counterparts, because of their inherently large, integrated, cross-dependent nature. Thus, a more popular algorithm, based upon the monolithic design, called the ***monolithic-modularized*** algorithm, has been implemented in OSes to allow for easier debugging, scalability and better performance over the standard monolithic approach. In a monolithic-modularized OS, the functionality is integrated into a single executable file that is made up of ***modules***, separate pieces of code reflecting various OS functionality. The embedded Linux operating system is an example of a monolithic-based OS, whose main modules are shown in Figure 9-4. The Jbed RTOS, MicroC/OS-II, and PDOS are all examples of embedded monolithic OSes.

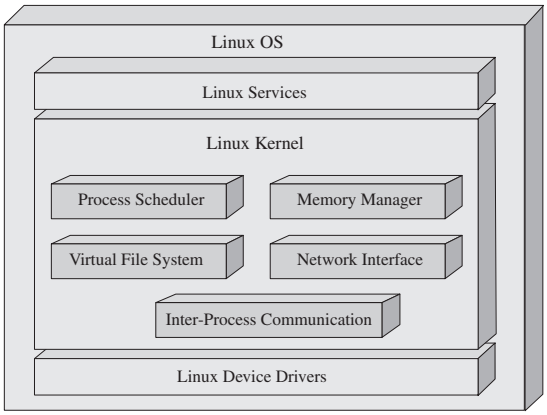


Figure 9-4: Linux OS block diagram

In the *layered* design, the OS is divided into hierarchical layers (0...N), where upper layers are dependent on the functionality provided by the lower layers. Like the monolithic design, layered OSes are a single large file that includes device drivers and middleware (see Figure 9-5). While the layered OS can be simpler to develop and maintain than a monolithic design, the APIs provided at each layer create additional overhead that can impact size and performance. DOS-C(FreeDOS), DOS/eRTOS, and VRTX are all examples of a layered OS.

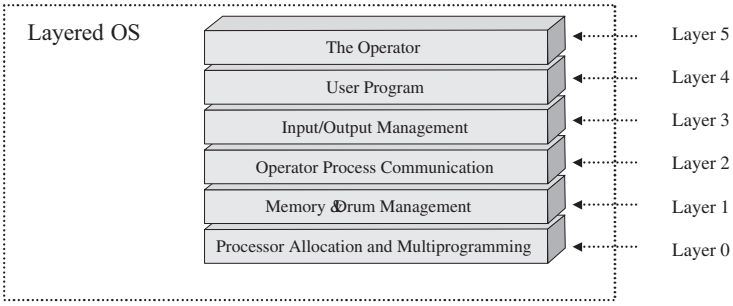


Figure 9-5: Layered OS block diagram

An OS that is stripped down to minimal functionality, commonly only process and memory management subunits as shown in Figure 9-6, is called a ***client-server*** OS, or a **microkernel**. (Note: a subclass of microkernels are stripped down even further to only process management functionality, and are commonly referred to as *nanokernels.*) The remaining functionality typical of other kernel algorithms is abstracted out of the kernel, while device drivers, for instance, are usually abstracted out of a microkernel entirely, as shown in Figure 9-6. A microkernel also typically differs in its process management implementation over other types of OSes. This is discussed in more detail in Section 9.1, Process Management: Intertask Communication and Synchronization.

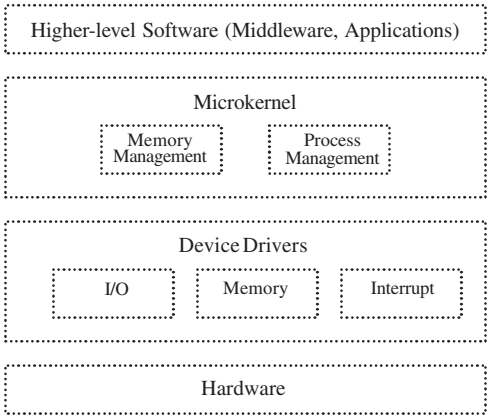


Figure 9-6: Microkernel-based OS block diagram

The microkernel OS is typically a more scalable (modular) and debuggable design, since additional components can be dynamically added in. It is also more secure since much of the functionality is now independent of the OS, and there is a separate memory space for client and server functionality. It is also easier to port to new architectures. However, this model may be slower than other OS architectures, such as the monolithic, because of the communication paradigm between the microkernel components and other "kernel-like" components. Overhead is also added when switching between the kernel and the other OS components and non-OS components (relative to layered and monolithic OS designs). Most of the off-the-shelf embedded OSes—and there are at least a hundred of them—have kernels that fall under the microkernel category, including: OS-9, C Executive, vxWorks, CMX-RTX, Nucleus Plus, and QNX.

9.1 What Is a Process?

To understand how OSes manage an embedded device's hardware and software resources, the reader must first understand how an OS views the system. An OS differentiates between a program and the executing of a program. A program is simply a *passive*, *static* sequence of instructions that could represent a system's hardware and software resources. The actual execution of a program is an *active*, *dynamic* event in which various properties change relative to time and the instruction being executed. A ***process*** (commonly referred to as a ***task*** in many embedded OSes) is created by an OS to encapsulate all the information that is involved in the executing of a program (i.e., stack, PC, the source code and data, etc.). This means that a program is only part of a task, as shown in Figure 9-7.

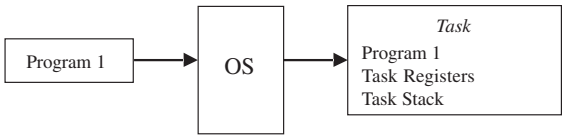


Figure 9-7: OS task

Embedded OSes manage all embedded software using tasks, and can either be ***unitasking*** or ***multitasking***. In unitasking OS environments, only one task can exist at any given time, whereas in a multitasking OS, multiple tasks are allowed to exist simultaneously. Unitasking OSes typically don't require as complex a task management facility as a multitasking OS. In a multitasking environment, the added complexity of allowing multiple existing tasks requires that each process remain independent of the others and not affect any other without the specific programming to do so. This multitasking model provides each process with more security, which is not needed in a unitasking environment. Multitasking can actually provide a more organized way for a complex embedded system to function. In a multitasking environment, system activities are divided up into simpler, separate components, or the same activities can be running in multiple processes simultaneously, as shown in Figure 9-8.

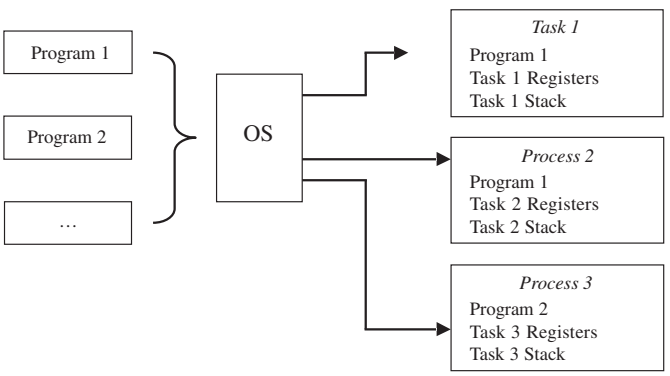


Figure 9-8: Multitasking OS

Some multitasking OSes also provide ***threads*** (*lightweight processes*) as an additional, alternative means for encapsulating an instance of a program. Threads are created within the context of a task (meaning a thread is bound to a task), and, depending on the OS, the task can own one or more threads. A thread is a sequential execution stream within its task. Unlike

tasks, which have their own independent memory spaces that are inaccessible to other tasks, threads of a task share the same resources (working directories, files, I/O devices, global data, address space, program code, etc.), but have their own PCs, stack, and scheduling information (PC, SP, stack, registers, etc.) to allow for the instructions they are executing to be scheduled independently. Since threads are created within the context of the same task and can share the same memory space, they can allow for simpler communication and coordination relative to tasks. This is because a task can contain at least one thread executing one program in one address space, or can contain many threads executing different portions of one program in one address space (see Figure 9-9), needing no intertask communication mechanisms. This is discussed in more detail at the end of section 9.1. Also, in the case of shared resources, multiple threads are typically less expensive than creating multiple tasks to do the same work.

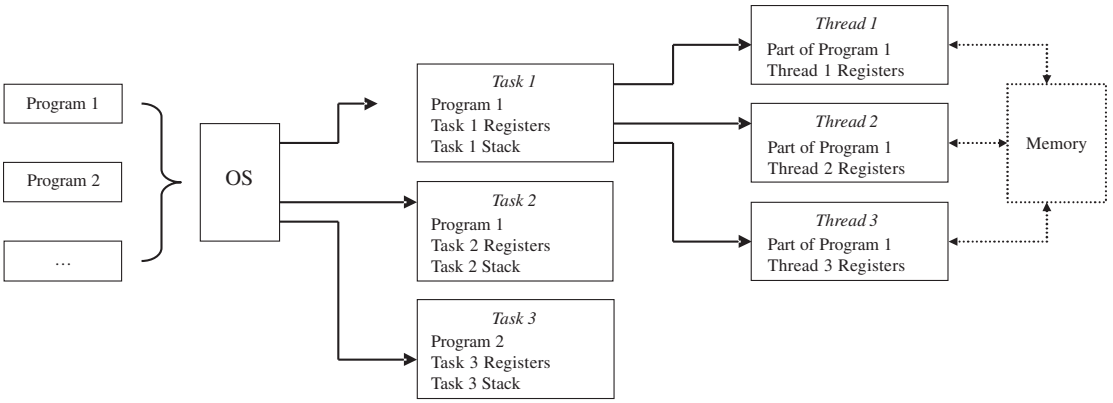


Figure 9-9: Tasks and threads

Usually, programmers define a separate task (or thread) for each of the system's distinct activities to simplify all the actions of that activity into a single stream of events, rather than a complex set of overlapping events. However, it is generally left up to the programmer as to how many tasks are used to represent a system's activity and, if threads are available, if and how they are used within the context of tasks.

DOS-C is an example of a unitasking embedded OS, whereas vxWorks (Wind River), embedded Linux (Timesys), and Jbed (Esmertec) are examples of multitasking OSes. Even within multitasking OSes, the designs can vary widely. vxWorks has one type of task, each of which implements one "thread of execution". Timesys Linux has two types of tasks, the linux fork and the periodic task, whereas Jbed provides six different types of tasks that run alongside threads: O*neshotTimer Task (*which is a task that is run only once), *PeriodicTimer Task* (a task that is run after a particular set time interval), *HarmonicEvent Task* (a task that runs alongside a periodic timer task), *JoinEvent Task* (a task that is set to run when an associated task completes), *InterruptEvent Task* (a task that is run when a hardware interrupt occurs), and the *UserEvent Task* (a task that is explicitly triggered by another task). More details on the different types of tasks are given in the next section.

9.2 Multitasking and Process Management

Multitasking OSes require an additional mechanism over unitasking OSes to manage and synchronize tasks that can exist simultaneously. This is because, even when an OS allows multiple tasks to coexist, one master processor on an embedded board can only execute one task or thread at any given time. As a result, multitasking embedded OSes must find some way of allocating each task a certain amount of time to use the master CPU, and switching the master processor between the various tasks. It is by accomplishing this through task ***implementation***, ***scheduling***, ***synchronization***, and ***inter-task communication*** mechanisms that an OS successfully gives the illusion of a single processor simultaneously running multiple tasks (see Figure 9-10).

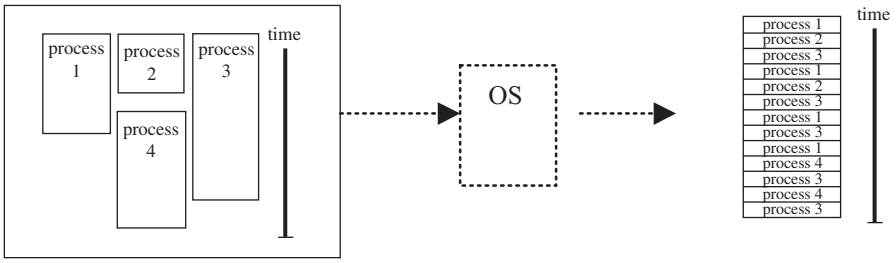


Figure 9-10: Interleaving tasks

9.2.1 Process Implementation

In multitasking embedded OSes, tasks are structured as a hierarchy of parent and child tasks, and when an embedded kernel starts up only one task exists (as shown in Figure 9-11). It is from this first task that all others are created (note: the first task is also created by the programmer in the system's initialization code, which will be discussed in more detail in Chapter 12).

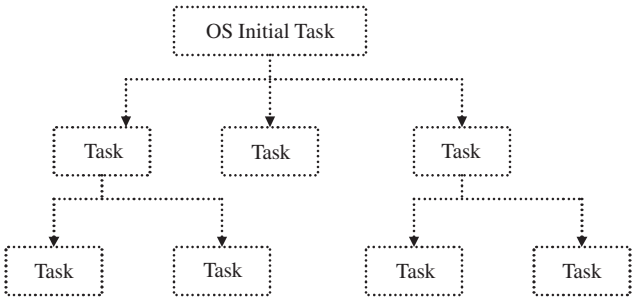
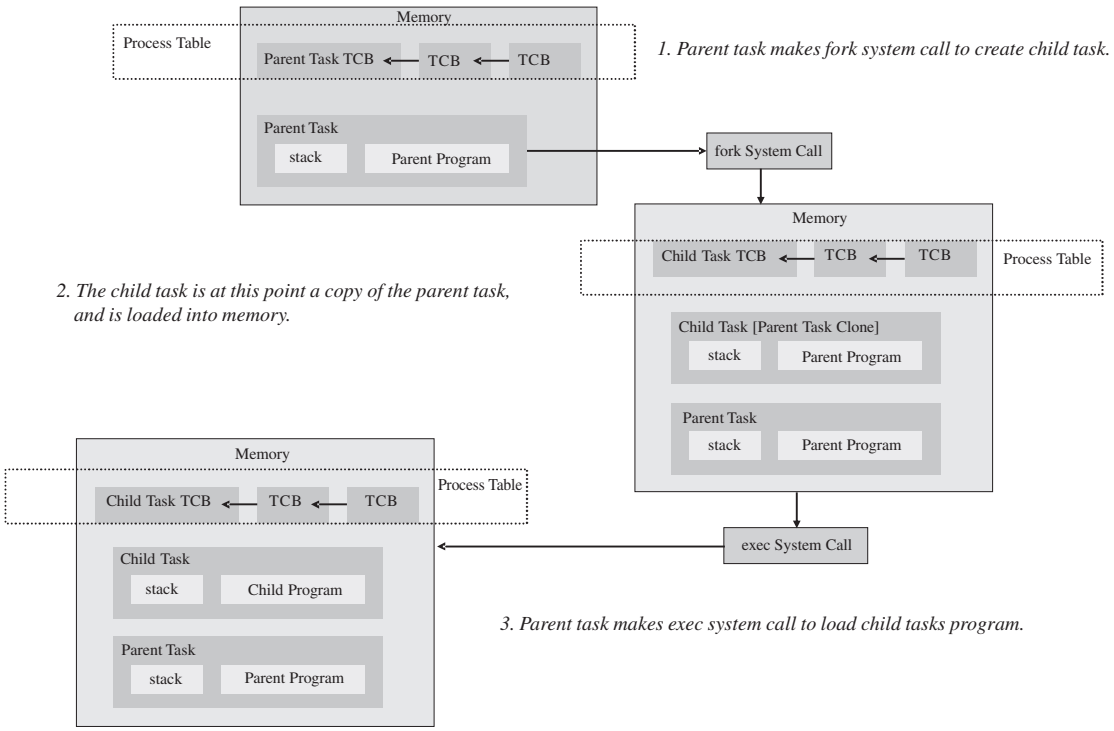


Figure 9-11: Task hierarchy

Task creation in embedded OSes is primarily based upon two models, ***fork/exec*** (which derived from the IEEE /ISO POSIX 1003.1 standard) and ***spawn*** (which is derived from fork/exec). Since the spawn model is based upon the fork/exec model, the methods of creating tasks under both models are similar. All tasks create their child tasks through fork/exec or

spawn system calls. After the system call, the OS gains control and creates the ***Task Control Block*** (TCB), also referred to as a *Process Control Block* (PCB) in some OSes, that contains OS control information, such as task ID, task state, task priority, and error status, and CPU context information, such as registers, for that particular task. At this point, memory is allocated for the new child task, including for its TCB, any parameters passed with the system call, and the code to be executed by the child task. After the task is set up to run, the system call returns and the OS releases control back to the main program.

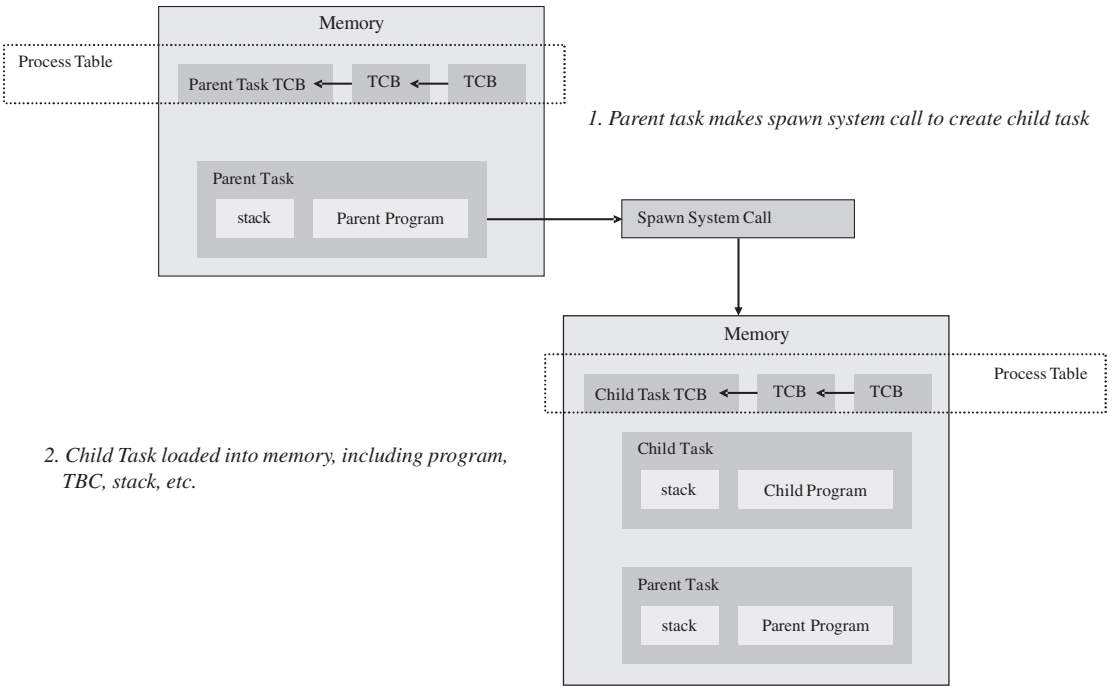
The main difference between the fork/exec and spawn models is how memory is allocated for the new child task. Under the fork/exec model, as shown in Figure 9-12, the "fork" call creates a copy of the parent task's memory space in what is allocated for the child task, thus allowing the child task to ***inherit*** various properties, such as program code and variables, from the parent task. Because the parent task's entire memory space is duplicated for the child task, two copies of the parent task's program code are in memory, one for the parent, and one belonging to the child. The "exec" call is used to explicitly remove from the child task's memory space any references to the parent's program and sets the new program code belonging to the child task to run.



<< Task creation based upon fork/exec involve 4 major steps >>

Figure 9-12: FORK/EXEC process creation

The spawn model, on the other hand, creates an entirely new address space for the child task. The spawn system call allows for the new program and arguments to be defined for the child task. This allows for the child task's program to be loaded and executed immediately at the time of its creation.



<< Task creation based upon spawn involve 2 major steps >>

Figure 9-13: Spawn process creation

Both process creation models have their strengths and drawbacks. Under the spawn approach, there are no duplicate memory spaces to be created and destroyed, and then new space allocated, as is the case with the fork/exec model. The advantages of the fork/exec model, however, include the efficiency gained by the child task inheriting properties from the parent task, and then having the flexibility to change the child task's environment afterwards. In Figures 9-1, 9-2, and 9-3, real-world embedded OSes are shown along with their process creation techniques.

EXAMPLE 9-1: Creating a task in vxWorks [9-1]

The two major steps of spawn task creation form the basis of creating tasks in vxWorks. The vxWorks system call "taskSpawn" is based upon the POSIX spawn model, and it is what creates, initializes, and activates a new (child) task.

```
int taskSpawn(
        {Task Name},
        {Task Priority 0-255, related to scheduling and will be discussed in the next section},
        {Task Options – VX_FP_TASK, execute with floating point coprocessor
              VX_PRIVATE_ENV, execute task with private environment
              VX_UNBREAKABLE, disable breakpoints for task
               VX_NO_STACK_FILL, do not fill task stack with 0xEE}
        {Stack Size}
        {Task address of entry point of program in memory– initial PC value}
        {Up to 10 arguments for task program entry routine})
```

After the spawn system call, an image of the child task (including TCB, stack, and program) is allocated into memory. Below is a pseudocode example of task creation in the vxWorks RTOS where a parent task "spawns" a child task software timer.

Task Creation vxWorks Pseudocode

```
// parent task that enables software timer
void parentTask(void)
{
…
if sampleSoftware Clock NOT running {

            /"newSWClkId" is a unique integer value assigned by kernel when task is created
            newSWClkId = taskSpawn ("sampleSoftwareClock", 255, VX_NO_STACK_FILL, 3000,
                                        (FUNCPTR) minuteClock, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
            ….
 }


//child task program Software Clock
void minuteClock (void) {
            integer seconds;

            while (softwareClock is RUNNING) {
                  seconds = 0;
                  while (seconds < 60) {
                        seconds = seconds+1;
            }
……
}
```

EXAMPLE 9-2: Jbed RTOS and task creation [9-2]

In Jbed, there is more than one way to create a task, because in Java there is more than one way to create a Java thread—and in Jbed, tasks are extensions of Java threads. One of the most common methods of creating a task in Jbed is through the "task" routines, one of which is:

```
public Task(long duration,
        long allowance,
        long deadline,
        RealtimeEvent event)
Throws AdmissionFailure
```

Task creation in Jbed is based upon a variation of the spawn model, called ***spawn threading***. Spawn threading is spawning, but typically with less overhead and with tasks sharing the same memory space. Below is a pseudocode example of task creation of a OneShot task, one of Jbed's six different types of tasks, in the Jbed RTOS where a parent task "spawns" a child task software timer that runs only one time.

Task Creation Jbed Pseudocode

```
// Define a class that implements the Runnable interface for the software clock
public class ChildTask implements Runnable{

        //child task program Software Clock
        public void run () {
            integer seconds;

            while (softwareClock is RUNNING) {
                seconds = 0;
                while (seconds < 60) {
                    seconds = seconds+1;
                }
                ……
            }
  }

// parent task that enables software timer
void parentTask(void)
{
…
if sampleSoftware Clock NOT running {

        try{
            DURATION,
            ALLOWANCE,
            DEADLINE,
            OneshotTimer );
        }catch( AdmissionFailure error ){
```

```
                    Print Error Message ( “Task creation failed” );
                }
}
….
 }
```

The creation and initialization of the Task object is the Jbed (Java) equivalent of a TCB. The task object, along with all objects in Jbed, are located in Jbed’s heap (in Java, there is only one heap for all objects). Each task in Jbed is also allocated its own stack to store primitive data types and object references.

EXAMPLE 9-3: Embedded Linux and fork/exec [9-3]

In embedded Linux, all process creation is based upon the fork/exec model:

```
int fork (void)                    void exec (…)
```

In Linux, a new “child” process can be created with the fork system call (shown above), which creates an almost identical copy of the parent process. What differentiates the parent task from the child is the process ID—the process ID of the child process is returned to the parent, whereas a value of “0” is what the child process believes its process ID to be.

```
#include <sys/types.h>
#include <unistd.h>

void program(void)
{

  processId child_processId;

     /* create a duplicate: child process */
     child_processId = fork();


      if (child_processId == -1) {
         ERROR;
      }
      else if (child_processId == 0) {
         run_childProcess();
      }
      else {
         run_parentParent();
      }
```

The exec function call can then be used to switch to the child’s program code.

```
int program (char* program, char** arg_list)
{
  processed  child_processId;
```

```
/* Duplicate this process */
child_processId = fork ();

if (child_pId != 0)

 /* This is the parent process */
 return child_processId;
 else
 {
 /* Execute PROGRAM, searching for it in the path */
 execvp (program, arg_list);

 /* execvp returns only if an error occurs */
 fprintf (stderr, "Error in execvp\n");
 abort ();
 }
}
```

Tasks can terminate for a number of different reasons, such as normal completion, hardware problems such as lack of memory, and software problems such as invalid instructions. After a task has been terminated, it must be removed from the system so that it doesn't waste resources, or even keep the system in limbo. In *deleting* tasks, an OS deallocates any memory allocated for the task (TCBs, variables, executed code, etc.). In the case of a parent task being deleted, all related child tasks are also deleted or moved under another parent, and any shared system resources are released.

Call	Description
exit()	Terminates the calling task and frees memory (task stacks and task control blocks only).
taskDelete()	Terminates a specified task and frees memory (task stacks and task control blocks only).*
taskSafe()	Protects the calling task from deletion.
taskUnsafe()	Undoes a taskSafe() (makes the calling task available for deletion).

* Memory that is allocated by the task during its execution is *not* freed when the task is terminated.

```
void vxWorksTaskDelete  (int taskId)
   {
      int localTaskId =taskIdFigure (taskId)  ;

    /* no such task ID */
    if (localTaskId =ERR     OR)
         printf (Error: ask not found.n);
    else if (localTaskId =0)
          printf (Error: The shell can't delete itself.n);
    else if (taskDelete (localTaskId) #OK
          printf (Error);
   }
```

Figure 9-14a: vxWorks and Spawn task deleted [9-4]

When a task is deleted in vxWorks, other tasks are not notified, and any resources, such as memory allocated to the task are not freed—it is the responsibility of the programmer to manage the deletion of tasks using the subroutines below.

In Linux, processes are deleted with the ***void exit(int status)*** system call, which deletes the process and removes any kernel references to process (updates flags, removes processes from queues, releases data structures, updates parent-child relationships, etc.). Under Linux, child processes of a deleted process become children of the main *init* parent process.

Figure 9-14b: Embedded Linux and fork/exec task deleted [9-3]

```
#include <stdio.h>
#include <stdlib.h>

main ()
{…
if (fork == 0)
  exit (10);
….
}
```

Because Jbed is based upon the Java model, a garbage collector is responsible for deleting a task and removing any unused code from memory once the task has stopped running. Jbed uses a non-blocking mark-and-sweep garbage collection algorithm which marks all objects still being used by the system and deletes (sweeps) all unmarked objects in memory.

In addition to creating and deleting tasks, an OS typically provides the ability to ***suspend*** a task (meaning temporarily blocking a task from executing) and ***resume*** a task (meaning any blocking of the task's ability to execute is removed). These two additional functions are provided by the OS to support task ***states***. A task's state is the activity (if any) that is going on with that task once it has been created, but has not been deleted. OSes usually define a task as being in one of three states:

- **Ready**: The process is ready to be executed at any time, but is waiting for permission to use the CPU.
- **Running**: The process has been given permission to use the CPU, and can execute.
- **Blocked** or **Waiting**: The process is waiting for some external event to occur before it can be "ready" to "run".

OSes usually implement separate READY and BLOCK/WAITING "queues" containing tasks (their TCBs) that are in the relative state (see Figure 9-15). Only one task at any one time can be in the RUNNING state, so no queue is needed for tasks in the RUNNING state.

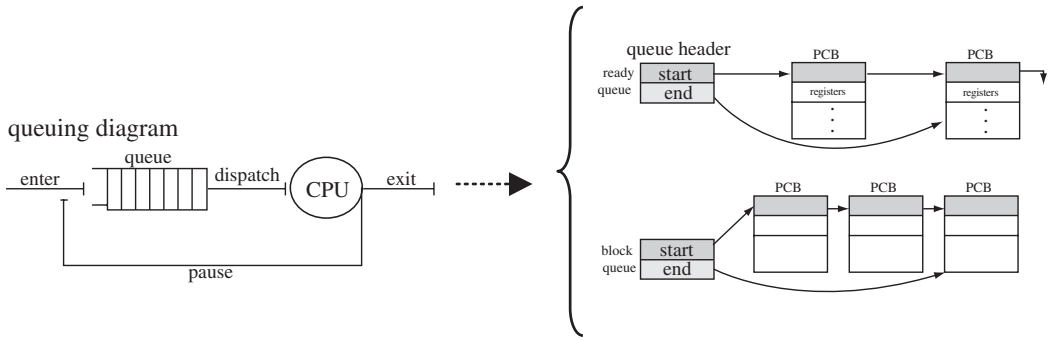


Figure 9-15: Task states and queues [9-4]

Based upon these three states (Ready, Blocked, and Running), most OSes have some process state transition model similar to the state diagram in Figure 9-16. In this diagram, the "New" state indicates a task that has been created, the "Exit" state is a task that has terminated (suspended or stopped running). The other three states are defined above (Ready, Running, and Blocked). The state transitions (according to Figure 9-16) are New → Ready (where a task has entered the ready queue and can be scheduled for running), Ready → Running (based on the kernel's scheduling algorithm, the task has been selected to run), Running → Ready (the task has finished its turn with the CPU, and is returned to the ready queue for the next time around), Running → Blocked (some event has occurred to move the task into the blocked queue, not to run until the event has occurred or been resolved), and Blocked → Ready (whatever blocked task was waiting for has occurred, and task is moved back to ready queue).

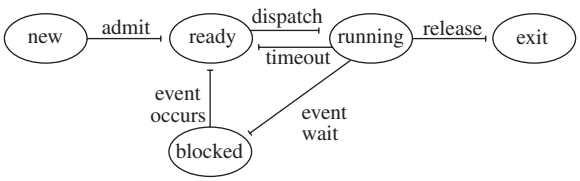


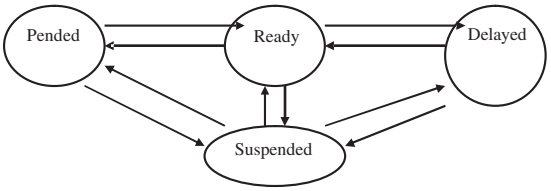
Figure 9-16: Task state diagram [9-2]

When a task is moved from one of the queues (READY or BLOCKED/WAITING) into the RUNNING state, it is called a ***context switch***. Examples 9-4, 9-5 and 9-6 give real-world examples of OSes and their state management schemes.

EXAMPLE 9-4: vxWorks Wind kernel and states [9-5]

Other than the RUNNING state, VxWorks implements nine variations of the READY and BLOCKED/WAITING states, as shown in the following table and state diagram.

State	Description
STATE + I	The state of the task with an inherited priority
READY	Task in READY state
DELAY	Task in BLOCKED state for a specific time period
SUSPEND	Task is BLOCKED usually used for debugging
DELAY + S	Task is in 2 states: DELAY & SUSPEND
PEND	Task in BLOCKED state due to a busy resource
PEND + S	Task is in 2 states: PEND & SUSPEND
PEND + T	Task is in PEND state with a timeout value
PEND + S + T	Task is in 2 states: PEND state with a timeout value and SUSPEND



This state diagram shows how a vxWorks task can switch between all of the various states.

Figure 9-17a1: State diagram for vxWorks tasks [9-5]

Under vxWorks, separate ready, pending, and delay state queues exist to store the TCB information of a task that is within that respective state (see Figure 9-17a2).

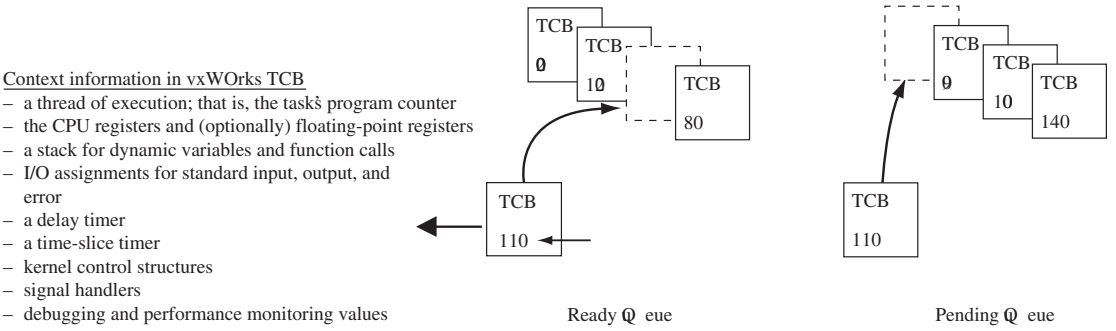


Figure 9-17a2: vxWorks tasks and queues [9-4]

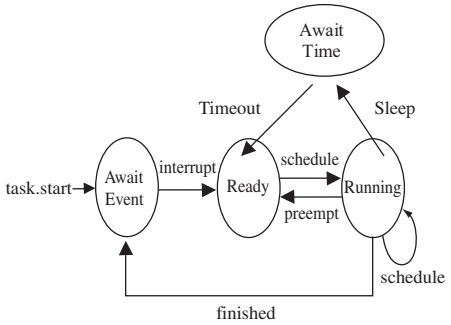
A task's TCB is modified and is moved from queue to queue when a context switch occurs. When the Wind kernel *context switches* between two tasks, the information of the task currently running is saved in its TCB, while the TCB information of the new task to be executed

is loaded for the CPU to begin executing. The Wind kernel contains two types of context switches: *synchronous*, which occurs when the running task blocks itself (through pending, delaying, or suspending), and *asynchronous*, which occurs when the running task is blocked due to an external interrupt.

EXAMPLE 9-5: Jbed kernel and states [9-6]

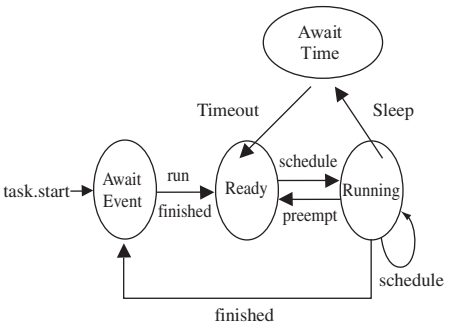
In Jbed, some states of tasks are related to the type of task, as shown in the table and state diagrams below. Jbed also uses separate queues to hold the task objects that are in the various states.

State	Description
RUNNING	For all types of tasks, task is currently executing
READY	For all types of tasks, task in READY state
STOP	In Oneshot Tasks, task has completed execution
AWAIT TIME	For all types of tasks, task in BLOCKED state for a specific time period
AWAIT EVENT	In Interrupt and Joined tasks, BLOCKED while waiting for some event to occur



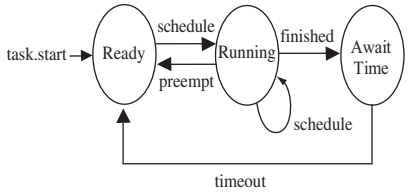
This state diagram shows some possible states for Interrupt tasks. Basically, an interrupt task is in an Await Event state until a hardware interrupt occurs – at which point the Jbed scheduler moves an Interrupt task into the Ready state to await its turn to run. At any time, the Joined Task can enter a timed waiting period.

Figure 9-17b1: State diagram for Jbed interrupt tasks [9-6]



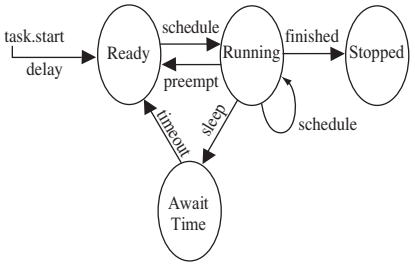
This state diagram shows some possible states for Joined tasks. Like the Interrupt task, the Joined task is in an Await Event state until an associated task has finished running – at which point the Jbed scheduler moves a Joined task into the Ready state to await its turn to run. At any time, the Joined Task can enter a timed waiting period.

Figure 9-17b2: State diagram for Jbed joined tasks [9-6]



This state diagram shows some possible states for Periodic tasks. A Periodic task runs continuously at certain intervals and gets moved into the Await Time state after every run to await that interval before being put into the ready state.

Figure 9-17b3: State diagram for periodic tasks [9-6]



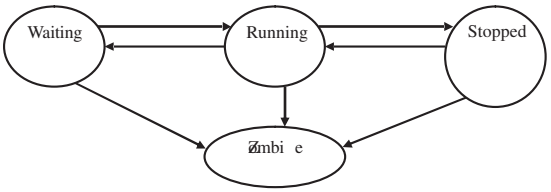
This state diagram shows some possible states for Oneshot tasks. A Oneshot task can either run once and then end (stop), or be blocked for a period of time before actually running.

Figure 9-17b4: State diagram for oneshot tasks [9-6]

EXAMPLE 9-6: Embedded Linux and states

In Linux, RUNNING combines the traditional READY and RUNNING states, while there are three variations of the BLOCKED state.

State	Description
RUNNING	Task is either in the RUNNING or READY state
WAITING	Task in BLOCKED state waiting for a specific resource or event
STOPPED	Task is BLOCKED, usually used for debugging
ZOMBIE	Task is BLOCKED and no longer needed



This state diagram shows how a Linux task can switch between all of the various states.

Figure 9-17c1: State diagram for Linux tasks [9-3]

Under Linux, a process's context information is saved in a PCB called the task_struct shown in Figure 9-17c2 below. Shown boldface in the figure is an entry in the task_struct containing a Linux process's state. In Linux there are separate queues that contain the task_struct (PCB) information for the process with that respective state.

```
 struct task_struct
{
                                        ....
// -1 unrunnable, 0 runnable, >0 stopped
  volatile long       state;
// number of clock ticks left to run in this scheduling slice, decremented by a timer.
  long                counter;
  // the process' static priority, only changed through well-known system calls like nice, POSIX.1b
 // sched_setparam, or 4.4BSD/SVR4 setpriority.
  long                priority;
  unsigned            long signal;
  //  bitmap of masked signals
  unsigned            long blocked;
// per process flags, defined below
  unsigned            long flags;
  int errno;
  // hardware debugging registers
  long                debugreg[8];
  struct exec_domain   *exec_domain;
  struct linux_binfmt  *binfmt;
  struct task_struct   *next_task, *prev_task;
  struct task_struct   *next_run,  *prev_run;
  unsigned long       saved_kernel_stack;
  unsigned long       kernel_stack_page;
  int                 exit_code, exit_signal;
    unsigned long       personality;
  int                 dumpable:1;
  int                 did_exec:1;
  int                 pid;
  int                 pgrp;
  int                 tty_old_pgrp;
  int                 session;
  // boolean value for session group leader
  int                 leader;
  int                 groups[NGROUPS];
  // pointers to (original) parent process, youngest child, younger sibling, older sibling, respectively.  (p->father
  // can be replaced with  p->p_pptr->pid)
  struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                      *p_ysptr, *p_osptr;
struct wait_queue    *wait_chldexit;
  unsigned short      uid,euid,suid,fsuid;
  unsigned short      gid,egid,sgid,fsgid;
  unsigned long       timeout;
// the scheduling policy, specifies which scheduling class the task belongs to, such as : SCHED_OTHER
//(traditional UNIX process), SCHED_FIFO(POSIX.1b FIFO realtime process - A FIFO realtime process will
//run until either a) it blocks on I/O, b) it explicitly yields the CPU or c) it is preempted by another realtime
//process with a higher p->rt_priority   value.) and SCHED_RR(POSIX round-robin realtime process –
//SCHED_RRis the same as SCHED_FIFQ except that when its timeslice expires it goes back to the end of the
//run queue).
  unsigned long       policy;
//realtime priority
  unsigned long       rt_priority;
  unsigned long       it_real_value, it_prof_value, it_virt_value;
  unsigned long       it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list   real_timer;
  long                utime, stime, cutime, cstime, start_time;
// mm fault and swap info: this can arguably be seen as either  mm-specific or thread-specific */
  unsigned long       min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
  int swappable:1;
  unsigned long       swap_address;
  // old value of maj_flt
  unsigned long       old_maj_flt;
  // page fault count of the last time
  unsigned long       dec_flt;
// number of pages to swap on next pass
  unsigned long       swap_cnt;
//limits
  struct rlimit       rlim[RLIM_NLIMITS];
  unsigned short      used_math;
  char                comm[16];
// file system info
  int                 link_count;
// NULL if no tty
  struct tty_struct    *tty;
// ipc stuff
  struct sem_undo      *semundo;
  struct sem_queue     *semsleeping;
// ldt for this task - used by Wine.  If NULL, default_ldt is used
  struct desc_struct *ldt;
// tss for this task
  struct thread_struct tss;
// filesystem information
  struct fs_struct     *fs;
// open file information
  struct files_struct  *files;
// memory management info
  struct mm_struct     *mm;
// signal handlers
  struct signal_struct *sig;
#ifdef __SMP__
  int                 processor;
  int                 last_processor;
  int                 lock_depth;    /* Lock depth.
                                        We can context switch in and out
                                        of holding a syscall kernel lock... */
#endif
}
```

Figure 9-17c2: Task structure [9-15]

9.2.2 Process Scheduling

In a multitasking system, a mechanism within an OS, called a ***scheduler*** (shown in Figure 9-18), is responsible for determining the order and the duration of tasks to run on the CPU. The scheduler selects which tasks will be in what states (ready, running, or blocked), as well as loading and saving the TCB information for each task. On some OSes the same scheduler allocates the CPU to a process that is loaded into memory and ready to run, while in other OSes a ***dispatcher*** (a separate scheduler) is responsible for the actual allocation of the CPU to the process.

There are many scheduling algorithms implemented in embedded OSes, and every design has its strengths and tradeoffs. The key factors that impact the effectiveness and performance of a scheduling algorithm include its ***response time*** (time for scheduler to make the context switch to a ready task and includes waiting time of task in ready queue), ***turnaround time*** (the time it takes for a process to complete running), ***overhead*** (the time and data needed to determine

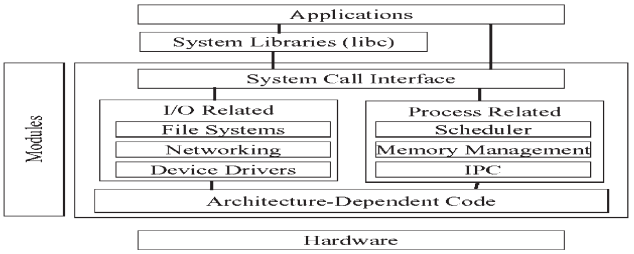


Figure 9-18: OS Block diagram and the scheduler [9-3]

which tasks will run next), and ***fairness*** (what are the determining factors as to which processes get to run). A scheduler needs to balance utilizing the system's resources—keeping the CPU, I/O, as busy as possible—with task ***throughput***, processing as many tasks as possible in a given amount of time. Especially in the case of fairness, the scheduler has to ensure that task ***starvation***, where a task never gets to run, doesn't occur when trying to achieve a maximum task throughput.

In the embedded OS market, scheduling algorithms implemented in embedded OSes typically fall under two approaches: ***non-preemptive*** and ***preemptive*** scheduling. Under non-preemptive scheduling, tasks are given control of the master CPU until they have finished execution, regardless of the length of time or the importance of the other tasks that are waiting. Scheduling algorithms based upon the non-preemptive approach include:

- **First-Come-First-Serve (FCFS)/ Run-To-Completion,** where tasks in the READY queue are executed in the order they entered the queue, and where these tasks are run until completion when they are READY to be run (see Figure 9-19). Here, non-preemptive means there is no BLOCKED queue in an FCFS scheduling design.

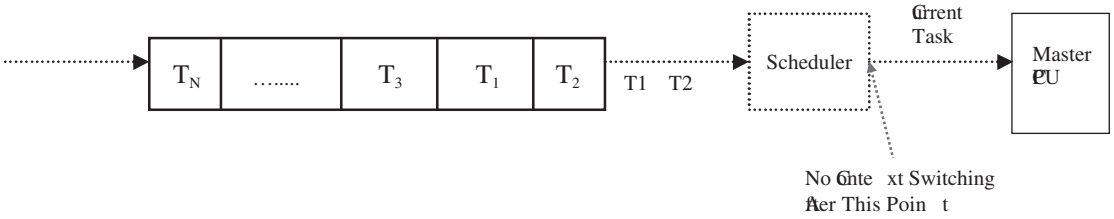


Figure 9-19: First-come-first-serve scheduling

The response time of a FCFS algorithm is typically slower than other algorithms (i.e., especially if longer processes are in front of the queue requiring that other processes wait their turn), which then becomes a fairness issue since short processes at the end of the queue get penalized for the longer ones in front. With this design, however, starvation is not possible.

- **Shortest Process Next (SPN)/Run-To-Completion**, where tasks in the READY queue are executed in the order in which the tasks with the shortest execution time are executed first. (see Figure 9-20).

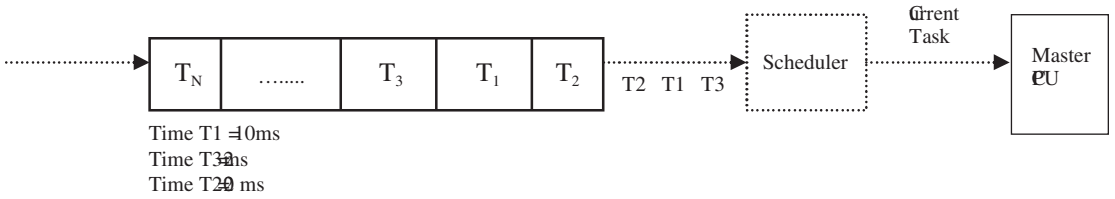


Figure 9-20: Shortest process next scheduling

The SPN algorithm has faster response times for shorter processes. However, then the longer processes are penalized by having to wait until all the shorter processes in the queue have run. In this scenario, starvation can occur to longer processes if the ready queue is continually filled with shorter processes. The overhead is higher than that of FCFS, since the calculation and storing of run times for the processes in the ready queue must occur.

- **Co-operative,** where the tasks themselves run until they tell the OS when they can be context switched (i.e., for I/O, etc.). This algorithm can be implemented with the FCFS or SPN algorithms, rather than the run-to-completion scenario, but starvation could still occur with SPN if shorter processes were designed not to "cooperate," for example.

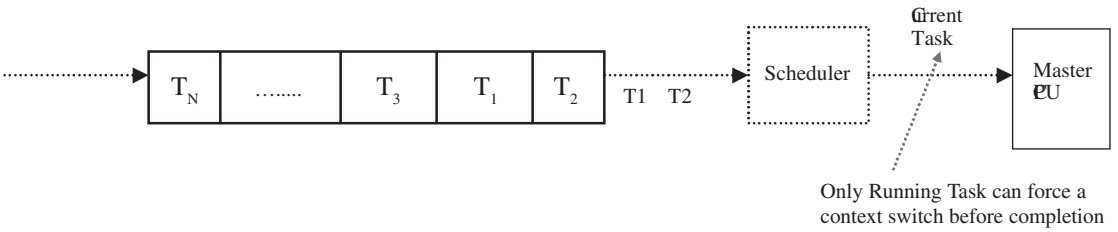


Figure 9-21: Co-operative scheduling

Non-preemptive algorithms can be riskier to support since an assumption must be made that no one task will execute in an infinite loop, shutting out all other tasks from the master CPU. However, OSes that support non-preemptive algorithms don't force a context-switch before a task is ready, and the overhead of saving and restoration of accurate task information when switching between tasks that have not finished execution is only an issue if the non-preemptive scheduler implements a co-operative scheduling mechanism. In ***preemptive scheduling***, on the other hand, the OS forces a context-switch on a task, whether or not a running task has completed executing or is cooperating with the context switch. Common scheduling algorithms based upon the preemptive approach include:

- **Round Robin/FIFO (First In, First Out) Scheduling**

  The Round Robin/FIFO algorithm implements a FIFO queue that stores *ready* processes (processes ready to be executed). Processes are added to the queue at the end of the queue, and are retrieved to be *run* from the start of the queue. In the FIFO system, all processes are treated equally regardless of their workload or interactivity. This is mainly due to the possibility of a single process maintaining control of the processor, never blocking to allow other processes to execute.

  Under round-robin scheduling, each process in the FIFO queue is allocated an equal ***time slice*** (the duration each process has to run), where an interrupt is generated at the end of each of these intervals to start the pre-emption process. (Note: scheduling algorithms that allocate time slices, are also referred to as ***time-sharing systems.***) The scheduler then takes turns rotating among the processes in the FIFO queue and executing the processes consecutively, starting at the beginning of the queue. New processes are added to the end of the FIFO queue, and if a process that is currently running isn't finished executing by the end of its allocated time slice, it is preempted and returned to the back of the queue to complete executing the next time its turn comes around. If a process finishes running before the end of its allocated time slice, the process voluntarily releases the processor, and the scheduler then assigns the next process of the FIFO queue to the processor (see Figure 9-22).

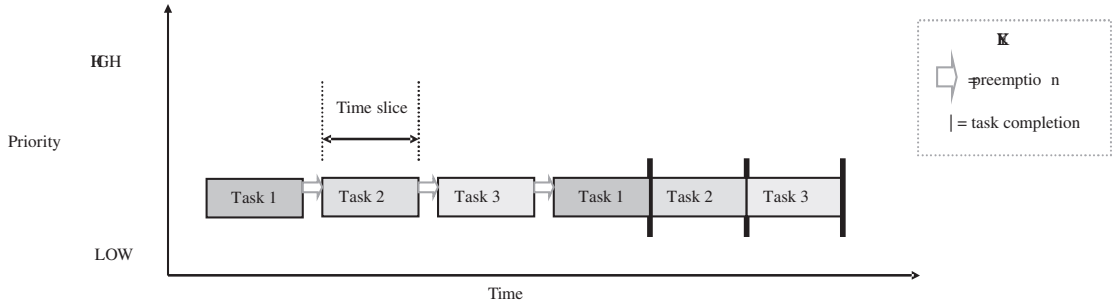


Figure 9-22: Round-robin/FIFO scheduling [9-7]

While Round Robin/FIFO scheduling ensures the equal treatment of processes, drawbacks surface when various processes have heavier workloads and are constantly preempted, thus creating more context switching overhead. Another issue occurs when processes in the queue are interacting with other processes (such as when waiting for the completion of another process for data), and are continuously preempted from completing any work until the other process of the queue has finished its run. The throughput depends on the time slice. If the time slice is too small, then there are many context switches, while too large a time slice isn't much different from a non-preemptive approach, like FCFS. Starvation is not possible with the round-robin implementation.

- **Priority (Preemptive) Scheduling**

  The ***priority preemptive scheduling*** algorithm differentiates between processes based upon their relative importance to each other and the system. Every process is assigned a priority, which acts as an indicator of orders of precedence within the system. The processes with the highest priority always preempt lower priority processes when they want to run, meaning a running task can be forced to block by the scheduler if a higher priority task becomes ready to run. Figure 9-23 shows three tasks (1, 2, 3—where task 1 is the lowest priority task and task 3 is the highest), and task 3 preempts task 2, and task 2 preempts task 1.

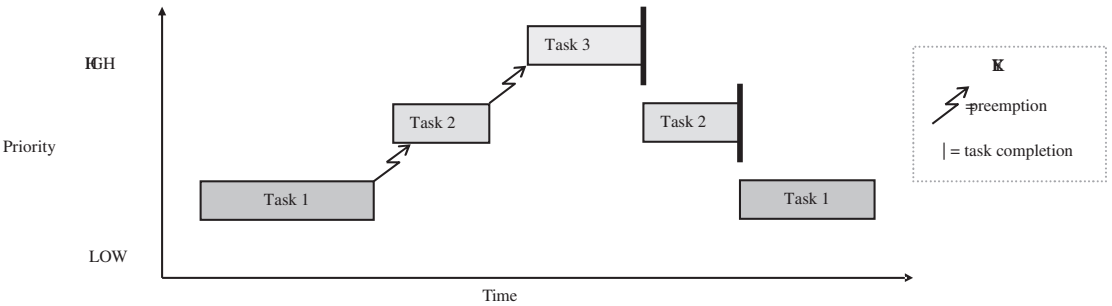


Figure 9-23: Preemptive priority scheduling [9-8]

While this scheduling method resolves some of the problems associated with round-robin/FIFO scheduling in dealing with processes that interact or have varying workloads, new problems can arise in priority scheduling including:

– *Process starvation*, where a continuous stream of high priority processes keep lower priority processes from ever running. Typically resolved by **aging** lower priority processes (as these processes spend more time on queue, increase their priority levels).

– *Priority inversion*, where higher priority processes may be blocked waiting for lower priority processes to execute, and processes with priorities in between have a higher priority in running, thus both the lower priority as well as higher priority processes don't run (see Figure 9-24).

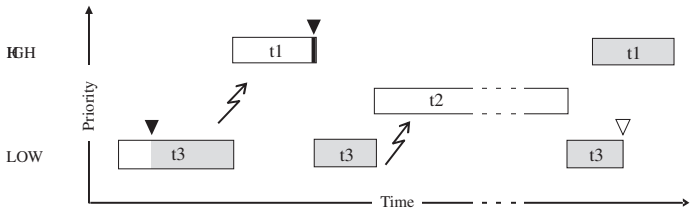


Figure 9-24: Priority inversion [9-8]

– how to *determine the priorities* of various processes. Typically, the more important the task, the higher the priority it should be assigned. For tasks that are equally important, one technique that can be used to assign task priorities is the ***Rate Monotonic Scheduling*** (RMS) scheme, in which tasks are assigned a priority based upon how often they execute within the system. The premise behind this model is that, given a preemptive scheduler and a set of tasks that are completely independent (no shared data or resources) and are run periodically (meaning run at regular time intervals), the more often a task is executed within this set, the higher its priority should be. The RMS Theorem says that if the above assumptions are met for a scheduler and a set of "n" tasks, all timing deadlines will be met if the inequality $\sum E_i/T_i <= n(2^{1/n} - 1)$ is verified, where

  i = periodic task
  n = number of periodic tasks
  T_i = the execution period of task i
  E_i = the worst-case execution time of task i
  E_i/T_i = the fraction of CPU time required to execute task i

So, given two tasks that have been prioritized according to their periods, where the shortest period task has been assigned the highest priority, the "$n(2^{1/n} - 1)$" portion of the inequality would equal approximately .828, meaning the CPU utilization of these tasks should not exceed about 82.8% in order to meet all hard deadlines. For 100 tasks that have been prioritized according to their periods, where the shorter period tasks have been assigned the higher priorities, CPU utilization of these tasks should not exceed approximately 69.6% (100 * (21/100 – 1)) in order to meet all deadlines.

Real-World Advice

To Benefit Most from a Fixed-Priority Preemptive OS

Algorithms for assigning priorities to OS tasks are typically classified as fixed-priority where tasks are assigned priorities at design time and do not change through the lifecycle of the task, dynamic-priority where priorities are assigned to tasks at run-time, or some combination of both algorithms. Many commercial OSes typically support only the fixed-priority algorithms, since it is the least complex scheme to implement. The key to utilizing the fixed-priority scheme is:

• to assign the priorities of tasks according to their periods, so that the shorter the periods, the higher the priorities.

• to assign priorities using a fixed-priority algorithm (like the Rate Monotonic Algorithm, the basis of RMS) to assign fixed priorities to tasks, and as a tool to quickly to determine if a set of tasks is schedulable.

• to understand that in the case when the inequality of a fixed-priority algorithm, like RMS, is not met, an analysis of the specific task set is required. RMS is a tool that allows for assuming that deadlines would be met in most cases if the total CPU utilization is below the limit

("most" cases meaning there are tasks that are not schedulable via any fixed-priority scheme). It is possible for a set of tasks to still be schedulable in spite of having a total CPU utilization above the limit given by the inequality. Thus, an analysis of each task's period and execution time needs to be done in order to determine if the set can meet required deadlines.

• to realize that a major constraint of fixed-priority scheduling is that it is not always possible to completely utilize the master CPU 100%. If the goal is 100% utilization of the CPU when using fixed priorities, then tasks should be assigned harmonic periods, meaning a task's period should be an exact multiple of all other tasks with shorter periods.

—Based on the article "Introduction to Rate Monotonic Scheduling" by Michael Barr
Embedded Systems Programming, February 2002

- **EDF (Earliest Deadline First)/Clock Driven Scheduling**

  As shown in Figure 9-25, the EDF/Clock Driven algorithm schedules priorities to processes according to three parameters: ***frequency*** (number of times process is run), ***deadline*** (when processes execution needs to be completed), and ***duration*** (time it takes to execute the process). While the EDF algorithm allows for timing constraints to be verified and enforced (basically guaranteed deadlines for all tasks), the difficulty is defining an exact duration for various processes. Usually, an average estimate is the best that can be done for each process.

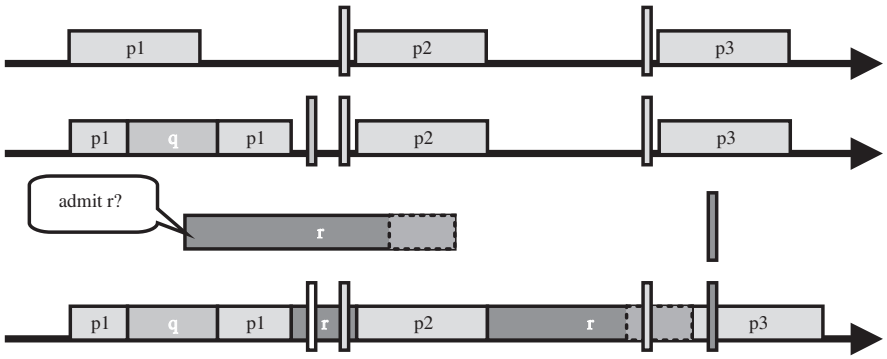


Figure 9-25: EDF scheduling [9-2]

Preemptive Scheduling and the Real-Time Operating System (RTOS)

One of the biggest differentiators between the scheduling algorithms implemented within embedded operating systems is whether the algorithm guarantees its tasks will meet execution time deadlines. If tasks always meet their deadlines (as shown in the first two graphs in Figure 9-26), and related execution times are predictable (deterministic), the OS is referred to as a ***Real-Time Operating System*** (RTOS).

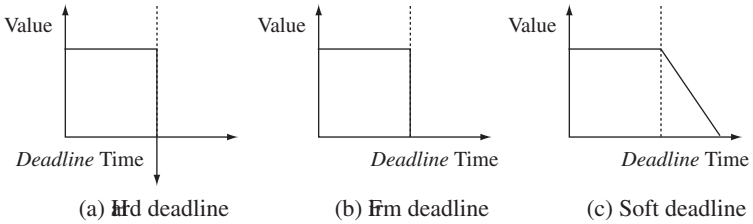


Figure 9-26: OSes and deadlines [9-4]

Preemptive scheduling must be one of the algorithms implemented within RTOS schedulers, since tasks with real-time requirements have to be allowed to preempt other tasks. RTOS schedulers also make use of their own array of *timers,* ultimately based upon the system clock, to manage and meet their hard deadlines.

Whether an RTOS or a nonreal-time OS in terms of scheduling, all will vary in their implemented scheduling schemes. For example, vxWorks (Wind River) is a priority-based and round-robin scheme, Jbed (Esmertec) is an EDF scheme, and Linux (Timesys) is a priority-based scheme. Examples 9-7, 9-8, and 9-9 examine further the scheduling algorithms incorporated into these embedded off-the-shelf operating systems.

EXAMPLE 9-7: vxWorks scheduling

The Wind scheduler is based upon both preemptive priority and round-robin real-time scheduling algorithms. As shown in Figure 9-27a1, round-robin scheduling can be teamed with preemptive priority scheduling to allow for tasks of the *same priority* to share the master processor, as well as allow higher priority tasks to preempt for the CPU.

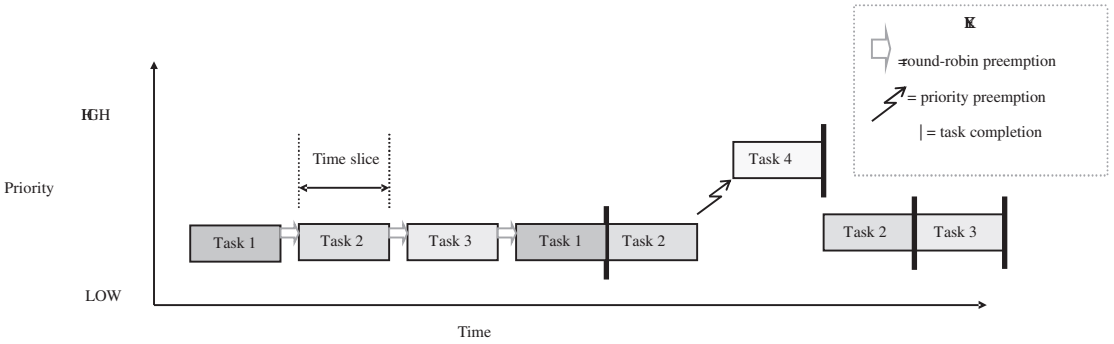


Figure 9-27a1: Preemptive priority scheduling augmented with round-robin scheduling [9-7]

Without round-robin scheduling, tasks of equal priority in vxWorks would never preempt each other, which can be a problem if a programmer designs one of these tasks to run in an infinite loop. However, the preemptive priority scheduling allows vxWorks its real-time capabilities, since tasks can be programmed never to miss a deadline by giving them the

higher priorities to preempt all other tasks. Tasks are assigned priorities via the "taskSpawn" command at the time of task creation:

```
int taskSpawn(
{Task Name},
{Task Priority 0-255, related to scheduling and will be discussed in the next section},
{Task Options – VX_FP_TASK, execute with floating point coprocessor
      VX_PRIVATE_ENV, execute task with private environment
      VX_UNBREAKABLE,   disable breakpoints for task
      VX_NO_STACK_FILL, do not fill task stack with 0xEE}
{Task address of entry point of program in memory– initial PC value}
{Up to 10 arguments for task program entry routine})
```

Example 9-8: Jbed and EDF scheduling

Under the Jbed RTOS, all six types of tasks have the three variables "duration", "allowance", and "deadline" when the task is created for the EDF scheduler to schedule all tasks, as shown in the method (java subroutine) calls below.

```
public Task(
            long duration,
            long allowance,
            long deadline,
            RealtimeEvent event)
   Throws AdmissionFailure
```

```
Public Task (java.lang.String name,
            long duration,
            long allowance,
            long deadline,
            RealtimeEvent event)
   Throws AdmissionFailure
```

```
Public Task (java.lang.Runnable target,
            java.lang.String name,
            long duration,
            long allowance,
            long deadline,
            RealtimeEvent event)
   Throws AdmissionFailure
```

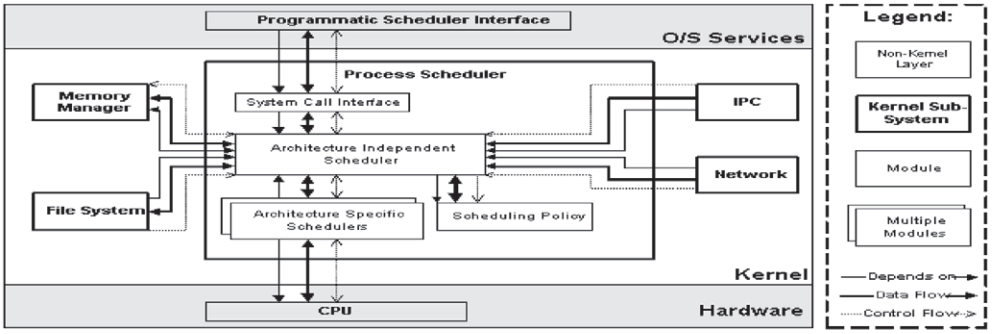


Figure 9-27b1: Embedded Linux block diagram [9-9]

EXAMPLE 9-9: TimeSys embedded Linux priority based scheduling

As shown in Figure 9-27b1, the embedded Linux kernel has a scheduler that is made up of four modules[9-9]:

- ***System call interface module***, which acts as the interface between user processes and any functionality explicitly exported by the kernel
- ***Scheduling policy module***, which determines which processes have access to the CPU.
- ***Architecture specific scheduler module***, which is an abstraction layer that interfaces with the hardware (i.e., communicating with CPU and the memory manager to suspend or resume processes)
- ***Architecture independent scheduler module***, which is an abstraction layer that interfaces between the scheduling policy module and the architecture specific module.

The scheduling policy module implements a "priority-based" scheduling algorithm. While most Linux kernels and their derivatives (2.2/2.4) are non-preemptable, have no rescheduling, and are not real-time, Timesys' Linux scheduler is priority-based, but has been modified to allow for real-time capabilities. Timesys has modified the traditional Linux's standard software timers, which are too coarsely grained to be suitable for use in most real-time applications because they rely on the kernel's jiffy timer, and implements high-resolution clocks and timers based on a hardware timer. The scheduler maintains a table listing all of the tasks within the entire system and any state information associated with the tasks. Under Linux, the total number of tasks allowed is only limited to the size of physical memory available. A dynamically allocated linked list of a task structure, whose fields that are relevant to scheduling are highlighted in Figure 9-27b2, represents all tasks in this table.

```
 struct task_struct
{                              ….
                                 // -1 unrunnable, 0 runnable, >0 stopped
  volatile long       state;

// number of clock ticks left to run in this scheduling slice, decremented
by a timer.
  long               counter;

  // the process' static priority, only changed through well-known system
calls like nice, POSIX.1b
 // sched_setparam, or 4.4BSD/SVR4 setpriority.
  long               priority;

  unsigned           long signal;

  //  bitmap of masked signals
  unsigned           long blocked;

 // per process flags, defined below
  unsigned           long flags;
  int errno;

  // hardware debugging registers
  long               debugreg[8];
  struct exec_domain   *exec_domain;
  struct linux_binfmt  *binfmt;
  struct task_struct   *next_task, *prev_task;
  struct task_struct   *next_run,  *prev_run;
  unsigned long       saved_kernel_stack;
  unsigned long       kernel_stack_page;
  int                exit_code, exit_signal;
   unsigned long       personality;
  int                dumpable:1;
  int                did_exec:1;
  int                pid;
  int                pgrp;
  int                tty_old_pgrp;
  int                session;
  // boolean value for session group leader
  int                leader;
  int                groups[NGROUPS];

  // pointers to (original) parent process, youngest child, younger sibling,
// older sibling, respectively.  (p->father   can be replaced with  p->p_pptr->pid)
  struct task_struct   *p_opptr, *p_pptr, *p_cptr,
                     *p_ysptr, *p_osptr;
  struct wait_queue    *wait_chldexit;
  unsigned short      uid,euid,suid,fsuid;
  unsigned short      gid,egid,sgid,fsgid;
  unsigned long       timeout;

// the scheduling policy, specifies which scheduling class the task belongs to,
// such as : SCHED_OTHER (traditional UNIX process), SCHED_FIFO
// (POSIX.1b FIFO realtime process - A FIFO realtime process will
//run until either a) it blocks on I/O, b) it explicitly yields the CPU or c) it is
// preempted by another realtime  process with a higher p->rt_priority value.)
// and SCHED_RR (POSIX round-robin realtime process –
//SCHED_RR is the same as SCHED_FIFO, except that when its timeslice
// expires it goes back to the end of the run queue).
 unsigned long       policy;

//realtime priority
  unsigned long     rt_priority;

  unsigned long       it_real_value, it_prof_value, it_virt_value;
  unsigned long       it_real_incr, it_prof_incr, it_virt_incr;
  struct timer_list   real_timer;
  long               utime, stime, cutime, cstime, start_time;

// mm fault and swap info: this can arguably be seen as either  mm-
specific or thread-specific */
  unsigned long       min_flt, maj_flt, nswap, cmin_flt, cmaj_flt,
cnswap;
  int swappable:1;
  unsigned long       swap_address;

  // old value of maj_flt
  unsigned long       old_maj_flt;

  // page fault count of the last time
  unsigned long       dec_flt;

// number of pages to swap on next pass
  unsigned long       swap_cnt;

//limits
  struct rlimit       rlim[RLIM_NLIMITS];
  unsigned short      used_math;
  char               comm[16];

// file system info
  int                link_count;

// NULL if no tty
  struct tty_struct    *tty;

// ipc stuff
  struct sem_undo      *semundo;
  struct sem_queue     *semsleeping;

// ldt for this task - used by Wine.  If NULL, default_ldt is used
  struct desc_struct *ldt;

// tss for this task
  struct thread_struct tss;

// filesystem information
  struct fs_struct     *fs;

// open file information
  struct files_struct  *files;

// memory management info
  struct mm_struct     *mm;

// signal handlers
  struct signal_struct *sig;
#ifdef __SMP__
  int                processor;
  int                last_processor;
  int                lock_depth;     /* Lock depth.
                                        We can context switch in and out
                                        of holding a syscall kernel lock... */
#endif
                                 …..
 }
```

Figure 9-27b2: Task structure [9-15]

After a process has been created in Linux, through the fork or fork/exec commands, for instance, its priority is set via the setpriority command.

```
int setpriority(int which, int who, int prio);
  which = PRIO_PROCESS, PRIO_PGRP, or PRIO_USER_
  who = interpreted relative to which
  prio = priority value in the range −20 to 20
```

9.2.3 Intertask Communication and Synchronization

Different tasks in an embedded system typically must share the same hardware and software resources, or may rely on each other in order to function correctly. For these reasons, embedded OSes provide different mechanisms that allow for tasks in a multitasking system to intercommunicate and synchronize their behavior so as to coordinate their functions, avoid problems, and allow tasks to run simultaneously in harmony.

Embedded OSes with multiple intercommunicating processes commonly implement interprocess communication (IPC) and synchronization algorithms based upon one or some combination of ***memory sharing***, ***message passing***, and ***signaling*** mechanisms.

With the ***shared data*** model shown in Figure 9-28, processes communicate via access to shared areas of memory in which variables modified by one process are accessible to all processes.

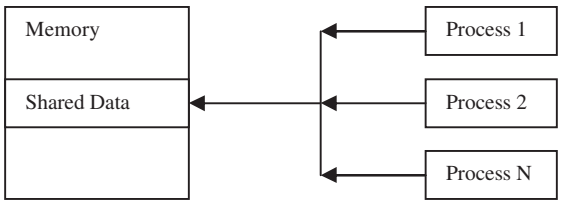


Figure 9-28: Memory sharing

While accessing shared data as a means to communicate is a simple approach, the major issue of ***race conditions*** can arise. A race condition occurs when a process that is accessing shared variables is preempted before completing a modification access, thus affecting the integrity of shared variables. To counter this issue, portions of processes that access shared data, called ***critical sections***, can be earmarked for ***mutual exclusion*** (or ***Mutex*** for short). Mutex mechanisms allow shared memory to be locked up by the process accessing it, giving that process exclusive access to shared data. Various mutual exclusion mechanisms can be implemented not only for coordinating access to shared memory, but for coordinating access to other shared system resources as well. Mutual exclusion techniques for *synchronizing* tasks that wish to concurrently access shared data can include:

- ***Processor assisted locks*** for tasks accessing shared data that are scheduled such that no other tasks can preempt them; the only other mechanisms that could force a context switch are interrupts. Disabling interrupts while executing code in the critical

section would avoid a race condition scenario if the interrupt handlers access the same data. Figure 9-29 demonstrates this processor-assisted lock of disabling interrupts as implemented in vxWorks.

VxWorks provides an interrupt locking and unlocking function for users to implement in tasks.

```
FuncA ()
        {
                int lock = intLock ();
                .
                . critical region that cannot be interrupted
                .
                intUnlock (lock);
        }
```

Figure 9-29: vxWorks processor assisted locks [9-10]

Another possible processor-assisted lock is the "test-and-set-instruction" mechanism (also referred to as the ***condition variable*** scheme). Under this mechanism, the setting and testing of a register flag (condition) is an atomic function, a process that cannot be interrupted, and this flag is tested by any process that wants to access a critical section.

In short, both the interrupt disabling and the condition variable type of locking schemes guarantee a process exclusive access to memory, where nothing can preempt the access to shared data and the system cannot respond to any other event for the duration of the access.

- ***Semaphores***, which can be used to lock access to shared memory (mutual exclusion), and also can be used to coordinate running processes with outside events (synchronization). The semaphore functions are *atomic* functions, and are usually invoked through system calls by the process. Example 9-10 demonstrates semaphores provided by vxWorks.

Example 9-10: vxWorks semaphores

VxWorks defines three types of semaphores:

1. *Binary* semaphores are binary (0 or 1) flags that can be set to be available or unavailable. Only the associated resource is affected by the mutual exclusion when a binary semaphore is used as a mutual exclusion mechanism (whereas processor assisted locks, for instance, can affect other unrelated resources within the system). A binary semaphore is initially set = 1 (full) to show the resource is available. Tasks check the binary semaphore of a resource when wanting access, and if available, then take the associated

semaphore when accessing a resource (setting the binary semaphore = 0), and then give it back when finishing with a resource (setting the binary semaphore = 1).

When a binary semaphore is used for task synchronization, it is initially set equal to 0 (empty), because it acts as an event other tasks are waiting for. Other tasks that need to run in a particular sequence then wait (block) for the binary semaphore to be equal to 1 (until the event occurs) to take the semaphore from the original task and set it back to 0. The vxWorks pseudocode example below demonstrates how binary semaphores can be used in vxWorks for task synchronization.

```
#include "vxWorks.h"
#include "semLib.h"
#include "arch/arch/ivarch.h" /* replace arch with architecture type */

SEM_ID syncSem; /* ID of sync semaphore */

init (int someIntNum)
{
 /* connect interrupt service routine */
 intConnect (INUM_TO_IVEC (someIntNum), eventInterruptSvcRout, 0);

 /* create semaphore */
 syncSem = semBCreate (SEM_Q_FIFO, SEM_EMPTY);

 /* spawn task used for synchronization. */
 taskSpawn ("sample", 100, 0, 20000, task1, 0,0,0,0,0,0,0,0,0,0);
}

task1 (void)
{
 ...
 semTake (syncSem, WAIT_FOREVER); /* wait for event to occur */
 printf ("task 1 got the semaphore\n");
 ... /* process event */
}

eventInterruptSvcRout (void)
{
 ...
 semGive (syncSem); /* let task 1 process event */
 ...
}
[9-4]
```

2. *Mutual Exclusion* semaphores are binary semaphores that can only be used for mutual exclusion issues that can arise within the vxWorks scheduling model, such as: priority inversion, deletion safety (insuring that tasks that are accessing a critical

section and blocking other tasks aren't unexpectedly deleted), and recursive access to resources. Below is a pseudocode example of a mutual exclusion semaphore used recursively by a task's subroutines.

```
/* Function A requires access to a resource which it acquires by taking
* mySem;
* Function A may also need to call function B, which also requires mySem:
*/
/* includes */
#include "vxWorks.h"
#include "semLib.h"
SEM_ID mySem;

/* Create a mutual-exclusion semaphore. */
init ()
{
  mySem = semMCreate (SEM_Q_PRIORITY);
}

funcA ()
{
  semTake (mySem, WAIT_FOREVER);
  printf ("funcA: Got mutual-exclusion semaphore\n");
  ...

  funcB ();
  ...
  semGive (mySem);
  printf ("funcA: Released mutual-exclusion semaphore\n");
}

funcB ()
{
  semTake (mySem, WAIT_FOREVER);
  printf ("funcB: Got mutual-exclusion semaphore\n");
  ...
  semGive (mySem);
  printf ("funcB: Releases mutual-exclusion semaphore\n");
}
[9-4]
```

3. *Counting* semaphores are positive integer counters with two related functions: incrementing and decrementing. Counting semaphores are typically used to manage multiple copies of resources. Tasks that need access to resources decrement the value of the semaphore, when tasks relinquish a resource, the value of the semaphore is incremented. When the semaphore reaches a value of "0", any task waiting for the related access is blocked until another task gives back the semaphore.

```
/* includes */
#include “vxWorks.h”
#include “semLib.h”
SEM_ID mySem;

/* Create a counting semaphore. */
init ()
{
  mySem = semCCreate (SEM_Q_FIFO,0);
}

……
[9-4]
```

On a final note, with mutual exclusion algorithms, only one process can have access to shared memory at any one time, basically having a *lock* on the memory accesses. If more than one process blocks waiting for their turn to access shared memory, and relying on data from each other, a ***deadlock*** can occur (such as priority inversion in priority based scheduling). Thus, embedded OSes have to be able to provide deadlock-avoidance mechanisms as well as deadlock-recovery mechanisms. As shown in the examples above, in vxWorks, semaphores are used to avoid and prevent deadlocks.

Intertask communication via ***message passing*** is an algorithm in which *messages* (made up of data bits) are sent via *message queues* between processes. The OS defines the protocols for process addressing and authentication to ensure that messages are delivered to processes reliably, as well as the number of messages that can go into a queue and the message sizes. As shown in Figure 9-30, under this scheme, OS tasks send messages to a message queue, or receive messages from a queue to communicate.

Microkernel-based OSes typically use the message passing scheme as their main synchronization mechanism. Example 9-11 demonstrates message passing in more detail, as implemented in vxWorks.

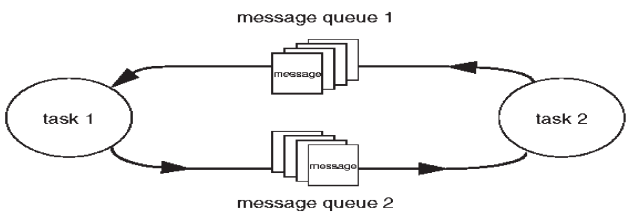


Figure 9-30: Message queues [9-4]

EXAMPLE 9-11: Message passing in vxWorks [9-4]

VxWorks allows for intertask communication via message passing queues to store data transmitted between different tasks or an ISR. VxWorks provides the programmer four system calls to allow for the development of this scheme:

Call	Description
msgQCreate( )	Allocates and initializes a message queue.
msgQDelete( )	Terminates and frees a message queue.
msgQSend( )	Sends a message to a message queue.
msgQReceive( )	Receives a message from a message queue.

These routines can then be used in an embedded application, as shown in the source code example below, to allow for tasks to intercommunicate:

```
/* In this example, task t1 creates the message queue and sends a message
* to task t2. Task t2 receives the message from the queue and simply
* displays the message.
*/

/* includes */
#include “vxWorks.h”
#include “msgQLib.h”

/* defines */
#define MAX_MSGS (10)
#define MAX_MSG_LEN (100)
MSG_Q_ID myMsgQId;

task2 (void)
{
  char msgBuf[MAX_MSG_LEN];
  /* get message from queue; if necessary wait until msg is available */
  if (msgQReceive(myMsgQId, msgBuf, MAX_MSG_LEN, WAIT_FOREVER) == ERROR)
  return (ERROR);
  /* display message */
  printf (“Message from task 1:\n%s\n”, msgBuf);
}

#define MESSAGE “Greetings from Task 1”
task1 (void)
{
  /* create message queue */
  if ((myMsgQId = msgQCreate (MAX_MSGS, MAX_MSG_LEN, MSG_Q_PRIORITY)) ==
NULL)
  return (ERROR);
  /* send a normal priority message, blocking if queue is full */
```

```
  if (msgQSend (myMsgQId, MESSAGE, sizeof (MESSAGE), WAIT_FOREVER,MSG_PRI_NOR-
MAL) ==
      ERROR)
  return (ERROR);
}
```

[9-4]

Signals and Interrupt Handling (Management) at the Kernel Level

Signals are indicators to a task that an asynchronous event has been generated by some external event (other processes, hardware on the board, timers, etc.) or some internal event (problems with the instructions being executed, etc.). When a task receives a signal, it suspends executing the current instruction stream and context switches to a signal handler (another set of instructions). The signal handler is typically executed within the task's context (stack) and runs in the place of the signaled task when it is the signaled task's turn to be scheduled to execute.

The wind *kernel supports two types of signal interface: UNIX BSD-style and POSIX-compatible signals.*

BSD 4.3	POSIX 1003.1
sigmask( )	sigemptyset( ), sigfillset( ), sigaddset( ), sigdelset( ), sigismember( )
sigblock( )	sigprocmask( )
sigsetmask( )	sigprocmask( )
pause( )	sigsuspend( )
sigvec( )	sigaction( )
(none)	sigpending( )
signal( )	signal( )
kill( )	kill( )

Figure 9-31: vxWorks signaling mechanism [9-4]

Signals are typically used for interrupt handling in an OS, because of their asynchronous nature. When a signal is raised, a resource's availability is unpredictable. However, signals can be used for general intertask communication, but are implemented so that the possibility of a signal handler blocking or a deadlock occurring is avoided. The other inter-task communication mechanisms (shared memory, message queues, etc.), along with signals, can be used for ISR-to-Task level communication, as well.

When signals are used as the OS abstraction for interrupts and the signal handling routine becomes analogous to an ISR, the OS manages the interrupt table, which contains the interrupt and information about its corresponding ISR, as well as provides a system call (subroutine) with parameters that that can be used by the programmer. At the same time, the OS protects the integrity of the interrupt table and ISRs, because this code is executed in kernel/

supervisor mode. The general process that occurs when a process receives a signal generated by an interrupt and an interrupt handler is called is shown in Figure 9-32.

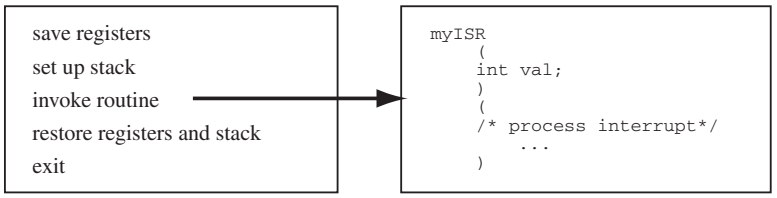


Figure 9-32: OS interrupt subroutine [9-4]

As mentioned in previous chapters, the architecture determines the interrupt model of an embedded system (that is, the number of interrupts and interrupt types). The interrupt device drivers initialize and provide access to interrupts for higher layer of software. The OS then provides the ***signal*** inter-process communication mechanism to allow for its processes to work with interrupts, as well as can provide various interrupt subroutines that abstracts out the device driver.

While all OSes have some sort of interrupt scheme, this will vary depending on the architecture they are running on, since architectures differ in their own interrupt schemes. Other variables include ***interrupt latency/response***, the time between the actual initiation of an interrupt and the execution of the ISR code, and ***interrupt recovery***, the time it takes to switch back to the interrupted task. Example 9-12 shows an interrupt scheme of a real-world embedded RTOS.

EXAMPLE 9-12: Interrupt handling in vxWorks

Except for architectures that do not allow for a separate interrupt stack (and thus the stack of the interrupted task is used), ISRs use the same interrupt stack, which is initialized and configured at system start-up, outside the context of the interrupting task. Table 9-1 summarizes the interrupt routines provided in vxWorks, along with a pseudocode example of using one of these routines.

Table 9-1: Interrupt routines in vxWorks [9-4]

Call	Description
intConnect( )	Connects a C routine to an interrupt vector.
intContext( )	Returns TRUE if called from interrupt level.
intCount( )	Gets the current interrupt nesting depth.
intLevelSet( )	Sets the processor interrupt mask level.
intLock( )	Disables interrupts.
intUnlock( )	Re-enables interrupts.
intVecBaseSet( )	Sets the vector base address.
intVecBaseGet( )	Gets the vector base address .
intVecSet( )	Sets an exception vector.
intVecGet( )	Gets an exception vector.

```
/* This routine initializes the serial driver, sets up interrupt vectors,
* and performs hardware initialization of the serial ports.
*/

void InitSerialPort (void)
{

initSerialPort();
(void) intConnect (INUM_TO_IVEC (INT_NUM_SCC), serialInt, 0);
.....
}
```

9.3 Memory Management

As mentioned earlier in this chapter, a kernel manages program code within an embedded system via tasks. The kernel must also have some system of loading and executing tasks within the system, since the CPU only executes task code that is in cache or RAM. With multiple tasks sharing the same memory space, an OS needs a security system mechanism to protect task code from other independent tasks. Also, since an OS must reside in the same memory space as the tasks it is managing, the protection mechanism needs to include managing its own code in memory and protecting it from the task code it is managing. It is these functions, and more, that are the responsibility of the memory management components of an OS. In general, a kernel's memory management responsibilities include:

- Managing the mapping between logical (physical) memory and task memory references.
- Determining which processes to load into the available memory space.
- Allocating and deallocating of memory for processes that make up the system.
- Supporting memory allocation and deallocation of code requests (within a process) such as the C language "alloc" and "dealloc" functions, or specific buffer allocation and deallocation routines.
- Tracking the memory usage of system components.
- Ensuring cache coherency (for systems with cache).
- Ensuring process memory protection.

As introduced in Chapters 5 and 8, physical memory is composed of two-dimensional arrays made up of cells addressed by a unique row and column, in which each cell can store 1 bit. Again, the OS treats memory as one large one-dimensional array, called a ***memory map.*** Either a hardware component integrated in the master CPU or on the board does the conversion between logical and physical addresses (such as an MMU), or it must be handled via the OS.

How OSes manage the logical memory space differs from OS to OS, but kernels generally run kernel code in a separate memory space from processes running higher level code (i.e., middleware and application layer code). Each of these memory spaces (*kernel* containing kernel code and *user* containing the higher-level processes) are managed differently. In fact, most OS processes typically run in one of two modes: ***kernel mode*** and ***user mode***, depending on the routines being executed. Kernel routines run in *kernel mode* (also referred to as *supervisor mode)*, in a different memory space and level than higher layers of software such as middleware or applications. Typically, these higher layers of software run in *user mode*, and can only access anything running in kernel mode via ***system calls***, the higher-level interfaces to the kernel's subroutines. The kernel manages memory for both itself and user processes.

9.3.1 User Memory Space

Because multiple processes are sharing the same physical memory when being loaded into RAM for processing, there also must be some protection mechanism so processes cannot inadvertently affect each other when being swapped in and out of a single physical memory space. These issues are typically resolved by the operating system through memory "swapping," where partitions of memory are *swapped* in and out of memory at run-time. The most common partitions of memory used in swapping are ***segments*** (fragmentation of processes from within) and ***pages*** (fragmentation of logical memory as a whole). Segmentation and paging not only simplify the swapping—memory allocation and deallocation—of tasks in memory, but allow for code reuse and memory protection, as well as providing the foundation for ***virtual memory***. Virtual memory is a mechanism managed by the OS to allow a device's limited memory space to be shared by multiple competing "user" tasks, in essence enlarging the device's actual physical memory space into a larger "virtual" memory space.

Segmentation

As mentioned in an earlier section of this chapter, a process encapsulates all the information that is involved in executing a program, including source code, stack, data, and so on. All of the different types of information within a process are divided into "logical" memory units of variable sizes, called ***segments***. A segment is a set of logical addresses containing the same type of information. Segment addresses are logical addresses that start at 0, and are made up of a ***segment number***, which indicates the base address of the segment, and a ***segment offset***, which defines the actual physical memory address. Segments are independently protected, meaning they have assigned accessibility characteristics, such as *shared* (where other processes can access that segment), Read-Only, or Read/Write.

Most OSes typically allow processes to have all or some combination of five types of information within segments: text (or code) segment, data segment, bss (block started by symbol) segment, stack segment, and the heap segment. A ***text*** segment is a memory space containing the source code. A ***data*** segment is a memory space containing the source code's initialized variables (data). A ***bss*** segment is a statically allocated memory space containing the source code's un-initialized variable (data). The data, text, and bss segments are all fixed in size at compile time, and are as such *static* segments; it is these three segments that typically are part of the *executable file*. Executable files can differ in what segments they are composed of, but in general they contain a header, and different sections that represent the types of segments, including name, permissions, and so on, where a segment can be made up of one or more sections. The OS creates a task's image by *memory mapping* the contents of the executable file, meaning loading and interpreting the segments (sections) reflected in the executable into memory. There are several executable file formats supported by embedded OSes, the most common including:

- ***ELF*** (Executable and Linking Format): UNIX-based, includes some combination of an ELF header, the program header table, the section header table, the ELF sections, and the ELF segments. Linux (Timesys) and vxWorks (WRS) are examples of OSes that support ELF.

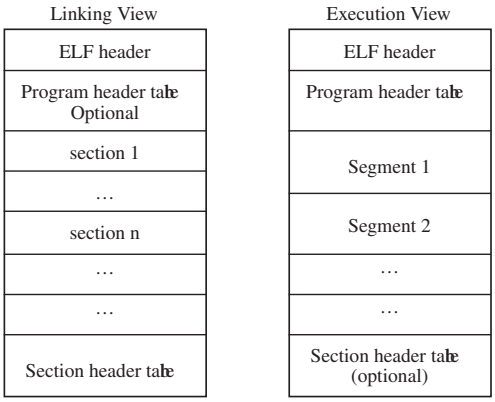


Figure 9-33: ELF executable file format [9-11]

- ***Class*** (Java Byte Code): A class file describes one java class in detail in the form of a stream of 8-bit bytes (hence the name "byte code"). Instead of segments, elements of the class file are called *items*. The Java class file format contains the class description, as well as, how that class is connected to other classes. The main components of a class file are a symbol table (with constants), declaration of fields, method implementations (code) and symbolic references (where other classes references are located). The Jbed RTOS is an example that supports the Java Byte Code format.

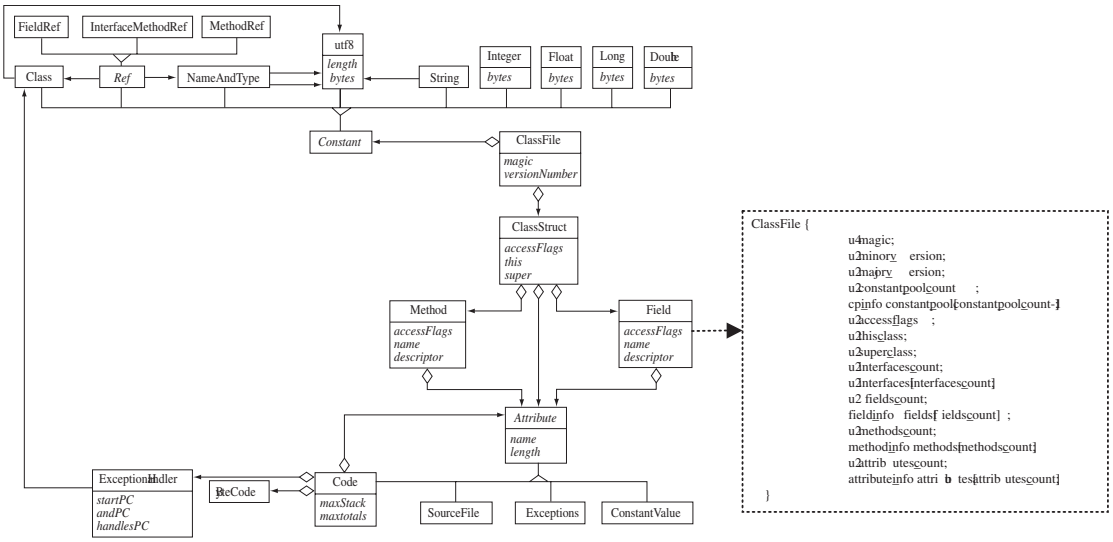


Figure 9-34: Class executable file format [9-12]

- ***COFF*** (Common Object File Format); A class file format which (among other things) defines an image file that contains file headers that include a file signature, COFF Header, an Optional Header, and also object files that contain only the COFF Header. Figure 9-35 shows an example of the information stored in a COFF header. WinCE[MS] is an example of an embedded OS that supports the COFF executable file format.

Offset	Size	Field	Description
0	2	Machine	Number identifying type of target machine.
2	2	Number of Sections	Number of sections; indicates size of the Section Table, which immediately follows the headers.
4	4	Time/Date Stamp	Time and date the file was created.
8	4	Pointer to Symbol	Offset, within the COFF file, of the symbol table.
12	4	Number of Symbols	Number of entries in the symbol table. This data can be used in locating the string table, which immediately follows the symbol table.
16	2	Optional Header	Size of the optional header, which is Size included for executable files but not object files. An object file should have a value of 0 here.
18	2	Characteristics	Flags indicating attributes of the file.

Figure 9-35: Class executable file format [9-13]

The *stack* and *heap* segments, on the other hand, are not fixed at compile time, and can change in size at runtime and so are *dynamic* allocation components. A ***stack*** segment is a section of memory that is structured as a LIFO (last in, first out) queue, where data is "Pushed" onto the stack, or "Popped" off of the stack (push and pop are the only two operations associated with a stack). Stacks are typically used as a simple and efficient method within a program for allocating and freeing memory for data that is predictable (i.e., local variables, parameter passing, etc.). In a stack, all used and freed memory space is located consecutively within the memory space. However, since "push" and "pop" are the only two operations associated with a stack, a stack can be limited in its uses.

A ***heap*** segment is a section of memory that can be allocated in blocks at runtime, and is typically set up as a free linked-list of memory fragments. It is here that a kernel's memory management facilities for allocating memory come into play to support the "malloc" C function (for example) or OS-specific buffer allocation functions. Typical memory allocation schemes include:

- *FF* (first fit) algorithm, where the list is scanned from the beginning for the first "hole" that is large enough.
- *NF* (next fit) where the list is scanned from where the last search ended for the next "hole" that is large enough.
- *BF* (best fit) where the entire list is searched for the hole that best fits the new data.
- *WF* (worst fit) which is placing data in the largest available "hole".
- *QF* (quick fit) where a list is kept of memory sizes and allocation is done from this information.

- *The buddy system*, where blocks are allocated in sizes of powers of 2. When a block is deallocated, it is then merged with contiguous blocks.

The method by which memory that is no longer needed within a heap is freed depends on the OS. Some OSes provide a garbage collector that automatically reclaims unused memory (garbage collection algorithms include generational, copying, and mark and sweep; see Figures 9-36a, b, and c). Other OSes require that the programmer explicitly free memory through a system call (i.e., in support of the "free" C function). With the latter technique, the programmer has to be aware of the potential problem of memory leaks, where memory is lost because it has been allocated but is no longer in use and has been forgotten, which is less likely to happen with a garbage collector.

Another problem occurs when allocated and freed memory cause memory fragmentation, where available memory in the heap is spread out in a number of holes, making it more difficult to allocate memory of the required size. In this case, a memory compaction algorithm must be implemented if the allocation/de-allocation algorithms causes a lot of fragmentation. This problem can be demonstrated by examining garbage collection algorithms.

The copying garbage collection algorithm works by copying referenced objects to a different part of memory, and then freeing up the original memory space. This algorithm uses a larger memory area to work, and usually cannot be interrupted during the copy (it blocks the systems). However, it does ensure that what memory is used, is used efficiently by compacting objects in the new memory space.

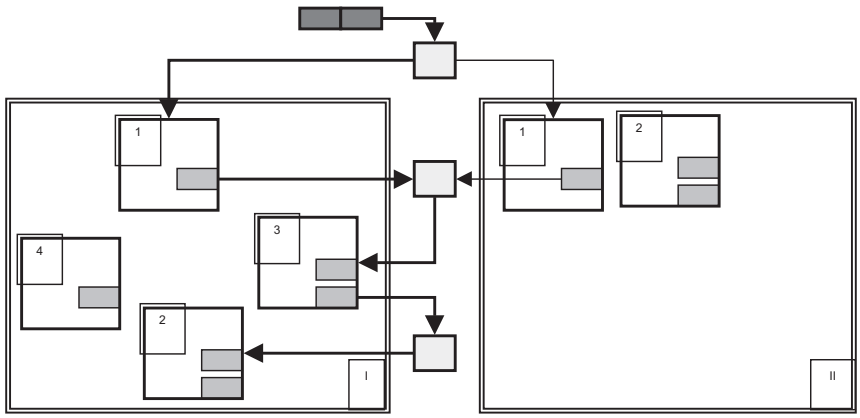
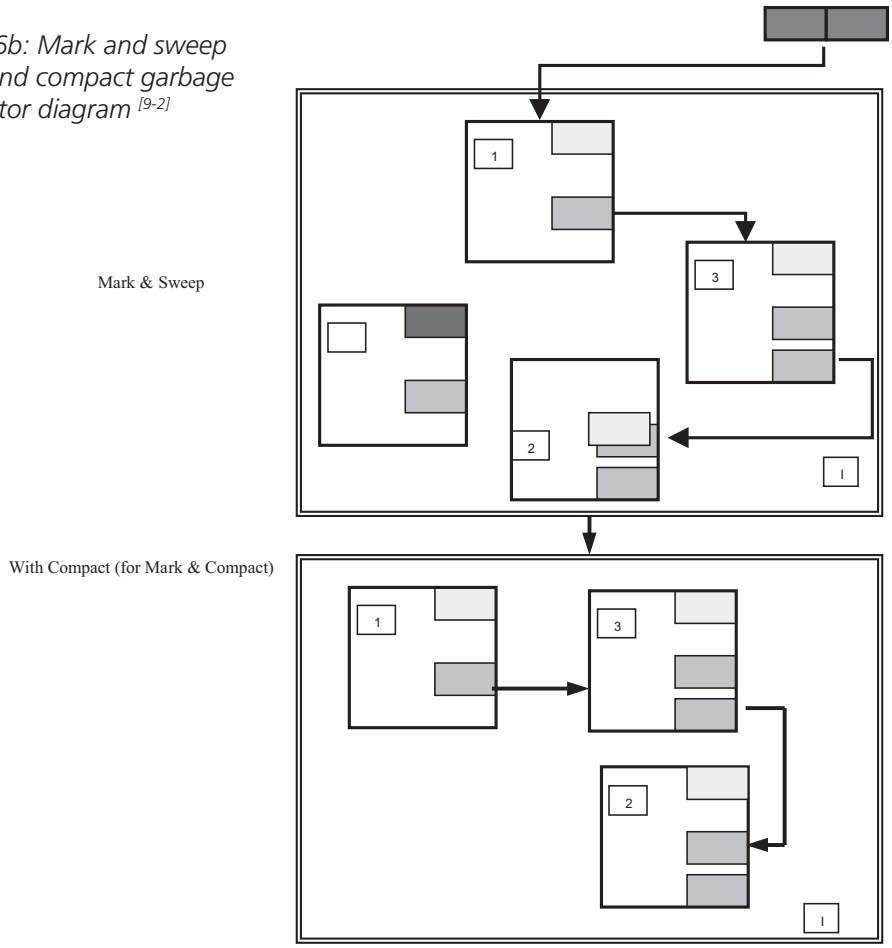


Figure 9-36a: Copying garbage collector diagram [9-2]

The mark and sweep garbage collection algorithm works by "marking" all objects that are used, and then "sweeping" (de-allocating) objects that are unmarked. This algorithm is usually nonblocking, so the system can interrupt the garbage collector to execute other functions when necessary. However, it doesn't compact memory the way a Copying garbage collector

would, leading to memory fragmentation with small, unusable holes possibly existing where deallocated objects used to exist. With a mark and sweep garbage collector, an additional memory compacting algorithm could be implemented making it a mark (sweep) and compact algorithm.

Figure 9-36b: Mark and sweep and mark and compact garbage collector diagram [9-2]



Finally, the generational garbage collection algorithm separates objects into groups, called *generations*, according to when they were allocated in memory. This algorithm assumes that most objects that are allocated are short-lived, thus copying or compacting the remaining objects with longer lifetimes is a waste of time. So, it is objects in the younger generation group that are cleaned up more frequently than objects in the older generation groups. Objects can also be moved from a younger generation to an older generation group. Each generational garbage collector also may employ different algorithms to de-allocate objects within each generational group, such as the copying algorithm or mark and sweep algorithms described above. Compaction algorithms would be needed in both generations to avoid fragmentation problems.

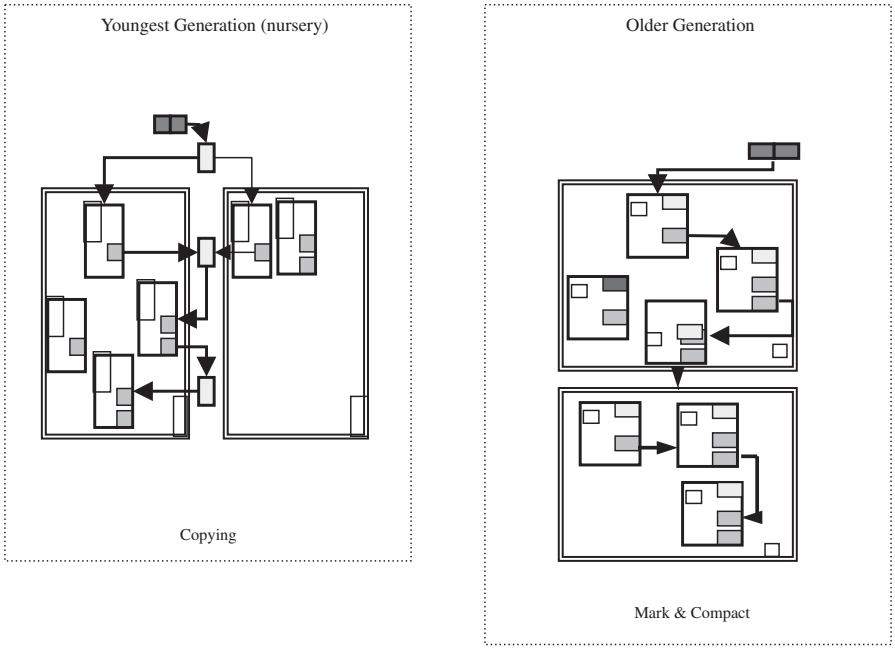


Figure 9-36c: Generational garbage collector diagram

Finally, heaps are typically used by a program when allocation and deletion of variables are unpredictable (linked lists, complex structures, etc.). However, heaps aren't as simple or as efficient as stacks. As mentioned, how memory in a heap is allocated and deallocated is typically affected by the programming language the OS is based upon, such as a C-based OS using "malloc" to allocate memory in a heap and "free" to deallocate memory or a Java-based OS having a garbage collector. Pseudocode examples 9-13, 9-14, and 9-15 demonstrate how heap space can be allocated and deallocated under various embedded OSes.

EXAMPLE 9-13: vxWorks memory management and segmentation

VxWorks tasks are made up of text, data, and bss static segments, as well as each task having its own stack.

The vxWorks system call "taskSpawn" is based upon the POSIX spawn model, and is what creates, initializes, and activates a new (child) task. After the spawn system call, an image of the child task (including TCB, stack, and program) is allocated into memory. In the pseudo-code below, the code itself is the text segment, data segments are any initialized variables, and the bss segments are the uninitialized variables (i.e., seconds,…). In the taskSpawn system call, the task stack size is 3000 bytes, and is not filled with 0xEE because of the VX_NO_ STACK_FILL parameter in the system call.

Task Creation vxWorks Pseudocode

```
// parent task that enables software timer
void parentTask(void)
{
…
if sampleSoftware Clock NOT running {

        /"newSWClkId" is a unique integer value assigned by kernel when task is created
        newSWClkId = taskSpawn ("sampleSoftwareClock", 255, VX_NO_STACK_FILL, 3000,
                                (FUNCPTR) minuteClock, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);
        ….
 }


//child task program Software Clock
void minuteClock (void) {
        integer seconds;

        while (softwareClock is RUNNING) {
                seconds = 0;
                while (seconds < 60) {
                        seconds = seconds+1;
        }
        ……
}
[9-4]
```

Heap space for vxWorks tasks is allocated by using the C-language malloc/new system calls to dynamically allocate memory. There is no garbage collector in vxWorks, so the programmer must deallocate memory manually via the free() system call.

```
/* The following code is an example of a driver that performs address
* translations. It attempts to allocate a cache-safe buffer, fill it, and
* then write it out to the device. It uses CACHE_DMA_FLUSH to make sure
* the data is current. The driver then reads in new data and uses
* CACHE_DMA_INVALIDATE to guarantee cache coherency. */
#include "vxWorks.h"
#include "cacheLib.h"
#include "myExample.h"
STATUS myDmaExample (void)
{
void * pMyBuf;
void * pPhysAddr;
/* allocate cache safe buffers if possible */
if ((pMyBuf = cacheDmaMalloc (MY_BUF_SIZE)) == NULL)
```

```
return (ERROR);
… fill buffer with useful information …
/* flush cache entry before data is written to device */
CACHE_DMA_FLUSH (pMyBuf, MY_BUF_SIZE);
/* convert virtual address to physical */
pPhysAddr = CACHE_DMA_VIRT_TO_PHYS (pMyBuf);
/* program device to read data from RAM */
myBufToDev (pPhysAddr);
… wait for DMA to complete …
… ready to read new data …
/* program device to write data to RAM */
myDevToBuf (pPhysAddr);
… wait for transfer to complete …
/* convert physical to virtual address */
pMyBuf = CACHE_DMA_PHYS_TO_VIRT (pPhysAddr);
/* invalidate buffer */
CACHE_DMA_INVALIDATE (pMyBuf, MY_BUF_SIZE);
… use data …
/* when done free memory */
if (cacheDmaFree (pMyBuf) == ERROR)
return (ERROR);
return (OK);
}
[9-4]
```

EXAMPLE 9-14: Jbed memory management and segmentation

In Java, memory is allocated in the Java heap via the "new" keyword (unlike the "malloc" in C, for example). However, there are a set of interfaces defined in some Java standards, called JNI or Java Native Interface, that allows for C and/or assembly code to be integrated within Java code, so in essence, the "malloc" is available if JNI is supported. For memory deallocation, as specified by the Java standard, is done via a garbage collector.

Jbed is a Java-based OS, and as such supports "new" for heap allocation.

```
public void CreateOneshotTask(){
  // Task execution time values
  final long DURATION = 100L; // run method takes < 100us
  final long ALLOWANCE = 0L; // no DurationOverflow handling
  final long DEADLINE = 1000L;// complete within 1000us
  Runnable target; // Task's executable code
  OneshotTimer taskType;
  Task task;

  // Create a Runnable object
  target = new MyTask();

  // Create oneshot tasktype with no delay
  taskType = new OneshotTimer( 0L );

  // Create the task
  try{
  task = new Task( target,
  DURATION, ALLOWANCE, DEADLINE,
  taskType );
 }catch( AdmissionFailure e ){
  System.out.println( "Task creation failed" );
 return;
 }
 [9-2]
```

Memory allocation in Java

Memory deallocation is handled automatically in the heap via a Jbed garbage collector based upon the mark and sweep algorithm (which is non-blocking and is what allows Jbed to be an RTOS). The GC can be run as a reoccurring task, or can be run by calling a "runGarbageCollector" method.

EXAMPLE 9-15: Linux memory management and segmentation

Linux processes are made up of text, data, and bss static segments, as well as each process has its own stack (which is created with the fork system call). Heap space for Linux tasks are allocated via the C-language malloc/new system calls to dynamically allocate memory. There is no garbage collector in Linux, so the programmer must deallocate memory manually via the free() system call.

```
void *mem_allocator (void *arg)
{
  int i;
  int thread_id = *(int *)arg;
  int start = POOL_SIZE * thread_id;
  int end = POOL_SIZE * (thread_id + 1);

  if(verbose_flag) {
   printf("Releaser %i works on memory pool %i to %i\n",
        thread_id, start, end);
   printf("Releaser %i started...\n", thread_id);
  }

  while(!done_flag) {

        /* find first NULL slot */
        for (i = start; i < end; ++i) {
          if (NULL == mem_pool[i]) {
                mem_pool[i] = malloc(1024);
                if (debug_flag)
                  printf("Allocate %i: slot %i\n",
                        thread_id, i);
                break;
          }
        }

  }
  pthread_exit(0);
}



void *mem_releaser(void *arg)
{
  int i;
  int loops = 0;
  int check_interval = 100;
  int thread_id = *(int *)arg;
  int start = POOL_SIZE * thread_id;
  int end = POOL_SIZE * (thread_id + 1);

  if(verbose_flag) {
   printf("Allocator %i works on memory pool %i to %i\n",
        thread_id, start, end);
   printf("Allocator %i started...\n", thread_id);
```

```
    }

    while(!done_flag) {

        /* find non-NULL slot */
        for (i = start; i < end; ++i) {
          if (NULL != mem_pool[i]) {
                    void *ptr = mem_pool[i];
                    mem_pool[i] = NULL;
                    free(ptr);
                    ++counters[thread_id];
                    if (debug_flag)
                      printf("Releaser %i: slot %i\n",
                              thread_id, i);
                    break;
          }
        }
        ++loops;
        if ( (0 == loops % check_interval) &&
            (elapsed_time(&begin) > run_time) ) {
                  done_flag = 1;
                  break;
        }
    }
    pthread_exit(0);
}
[9-3]
```

Paging and Virtual Memory

Either with or without segmentation, some OSes divide logical memory into some number of fixed-size partitions, called ***blocks***, ***frames***, ***pages*** or ***some combination of a few or all of these***. For example, with OSes that divide memory into frames, the logical address is compromised of a frame number and offset. The user memory space can then, also, be divided into pages, where page sizes are typically equal to *frame* sizes.

When a process is loaded in its entirety into memory (in the form of pages), its pages may not be located within a contiguous set of frames. Every process has an associated process table that tracks its pages, and each page's corresponding frames in memory. The logical address spaces generated are unique for each process, even though multiple processes share the same physical memory space. Logical address spaces are typically made up of a page-frame number, which indicates the start of that page, and an offset of an actual memory location within that page. In essence, the logical address is the sum of the page number and the offset.

An OS may start by *prepaging*, or loading the pages needed to get started, and then implementing the scheme of ***demand paging*** where processes have no pages in memory, and pages

are only loaded into RAM when a ***page fault*** (an error occurring when attempting to access a page not in RAM) occurs. When a page fault occurs, the OS takes over and loads the needed page into memory, updates page tables, and then the instruction that triggered the page fault in the first place is re-executed. This scheme is based upon Knuth's Locality of Reference theory, which estimates that 90% of a system's time is spent on processing just 10% of code.

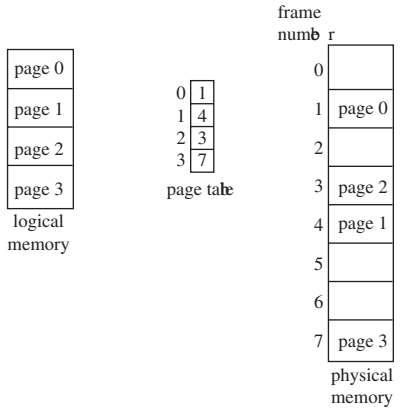


Figure 9-37: Paging [9-3]

Dividing up logical memory into pages aids the OS in more easily managing tasks being relocated in and out of various types of memory in the memory hierarchy, a process called ***swapping***. Common page selection and replacement schemes to determine which pages are swapped include:

- ***Optimal***, using future reference time, swapping out pages that won't be used in the near future.
- ***Least Recently Used*** (LRU), which swaps out pages that have been used the least recently.
- ***FIFO*** (First-In-First-Out), which as its name implies, swaps out the pages that are the oldest (regardless of how often it is accessed) in the system. While a simpler algorithm then LRU, FIFO is much less efficient.
- ***Not Recently Used*** (NRU), swaps out pages that were not used within a certain time period.
- ***Second Chance***, FIFO scheme with a reference bit, if "0" will be swapped out (a reference bit is set to "1" when access occurs, and reset to "0" after the check).
- ***Clock Paging***, pages replaced according to clock (how long they have been in memory), in clock order, if they haven't been accessed (a reference bit is set to "1" when access occurs, and reset to "0" after the check).

While every OS has its own swap algorithm, all are trying to reduce the possibility of ***thrashing***, a situation in which a system's resources are drained by the OS constantly swapping in and out data from memory. To avoid thrashing, a kernel may implement a ***working set*** model, which keeps a fixed number of pages of a process in memory at all times. Which pages (and the number of pages) that comprise this working set depends on the OS, but typically it is the pages accessed most recently. A kernel that wants to prepage a process also needs to have a working set defined for that process before the process's pages are swapped into memory.

Virtual Memory

Virtual memory is typically implemented via demand segmentation (fragmentation of processes from within, as discussed in a previous section) and/or demand paging (fragmentation of logical user memory as a whole) memory fragmentation techniques. When virtual memory is implemented via these "demand" techniques, it means that only the pages and/or segments that are currently in use are loaded into RAM.

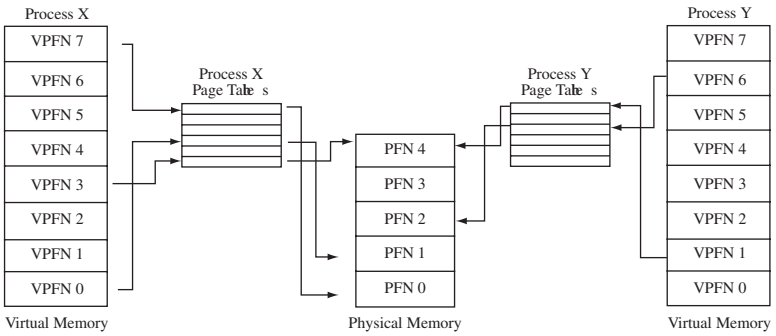


Figure 9-38: Virtual memory [9-3]

As shown in Figure 9-38, in a virtual memory system, the OS generates ***virtual*** addresses based on the logical addresses, and maintains ***tables*** for the sets of logical addresses into virtual addresses conversions (on some processors table entries are cached into TLBs, see Chapters 4 and 5 for more on MMUs and TLBs). The OS (along with the hardware) then can end up managing more than one different address space for each process (the physical, logical, and virtual). In short, the software being managed by the OS views memory as one continuous memory space, whereas the kernel actually manages memory as several fragmented pieces which can be segmented and paged, segmented and unpaged, unsegmented and paged, or unsegmented and unpaged.

9.3.2 Kernel Memory Space

The kernel's memory space is the portion of memory in which the kernel code is located, some of which is accessed via system calls by higher-level software processes, and is where the CPU executes this code from. Code located in the kernel memory space includes required IPC mechanisms, such as those for message passing queues. Another example is when tasks are creating some type of fork/exec or spawn system calls. After the task creation system call, the OS gains control and creates the ***Task Control Block*** (TCB), also referred to as a *Process Control Block* (PCB) in some OSes, within the kernel's memory space that contains OS control information and CPU context information for that particular task. Ultimately, what is managed in the kernel memory space, as opposed to in the user space, is determined by the hardware, as well as the actual algorithms implemented within the OS kernel.

As previously mentioned, software running in user mode can only access anything running in kernel mode via ***system calls***. System calls are the higher-level (user mode) interfaces to the

kernel's subroutines (running in kernel mode). Parameters associated with system calls that need to be passed between the OS and the system callee running in user mode are then passed via registers, a stack, or in the main memory heap. The types of system calls typically fall under the types of functions being supported by the OS, so they include file systems management (i.e., opening/modifying files), process management (i.e., starting/stopping processes), I/O communications, and so on. In short, where an OS running in kernel mode views what is running in user mode as processes, software running in user mode views and defines an OS by its system calls.

9.4 I/O and File System Management

Some embedded OSes provide memory management support for a temporary or permanent file system storage scheme on various memory devices, such as Flash, RAM, or hard disk. File systems are essentially a collection of files along with their management protocols (see Table 9-2). File system algorithms are middleware and/or application software that is ***mounted*** (installed) at some mount point (location) in the storage device.

Table 9-2: Middleware file system standards

File System	Summary
FAT32 (File Allocation Table)	Where memory is divided into the smallest unit possible (called sectors). A group of sectors is called a cluster. An OS assigns a unique number to each cluster, and tracks which files use which clusters. FAT32 supports 32-bit addressing of clusters, as well as smaller cluster sizes than that of the FAT predecessors (FAT, FAT16, etc.)
NFS (Network File System)	Based on RPC (Remote Procedure Call) and XDR (Extended Data Representation), NFS was developed to allow external devices to mount a partition on a system as if it were in local memory. This allows for fast, seamless sharing of files across a network.
FFS (Flash File System)	Designed for Flash memory.
DosFS	Designed for real-time use of block devices (disks) and compatible with the MS-DOS file system.
RawFS	Provides a simple *raw file system* that essentially treats an entire disk as a single large file.
TapeFS	Designed for tape devices that do not use a standard file or directory structure on tape. Essentially treats the tape volume as a raw device in which the entire volume is a large file.
CdromFS	Allows applications to read data from CD-ROMs formatted according to the ISO 9660 standard file system.

In relation to file systems, a kernel typically provides file system management mechanisms for, at the very least:

- *Mapping* files onto secondary storage, Flash, or RAM (for instance).
- Supporting the primitives for manipulating files and directories.
  - *File Definitions and Attributes*: Naming Protocol, Types (i.e., executable, object, source, multimedia, etc.), Sizes, Access Protection (Read, Write, Execute, Append, Delete, etc.), Ownership, and so on.
  - *File Operations*: Create, Delete, Read, Write, Open, Close, and so on.
  - *File Access Methods*: Sequential, Direct, and so on.
  - *Directory Access, Creation* and *Deletion*.

OSes vary in terms of the primitives used for manipulating files (i.e., naming, data structures, file types, attributes, operations, etc.), what memory devices files can be mapped to, and what file systems are supported. Most OSes use their standard I/O interface between the file system and the memory device drivers. This allows for one or more file systems to operate in conjunction with the operating system.

I/O Management in embedded OSes provides an additional abstraction layer (to higher level software) away from the system's hardware and device drivers. An OS provides a uniform interface for I/O devices that perform a wide variety of functions via the available kernel system calls, providing protection to I/O devices since user processes can only access I/O via these system calls, and managing a fair and efficient I/O sharing scheme among the multiple processes. An OS also needs to manage synchronous and asynchronous communication coming from I/O to its processes, in essence be event-driven by responding to requests from both sides (the higher level processes and low-level hardware), and manage the data transfers. In order to accomplish these goals, an OS's I/O management scheme is typically made up of a generic device-driver interface both to user processes and device drivers, as well as some type of buffer-caching mechanism.

Device driver code controls a board's I/O hardware. In order to manage I/O, an OS may require all device driver code to contain a specific set of functions, such as startup, shutdown, enable, disable, and so on. A kernel then manages I/O devices, and in some OSes file systems as well, as "black boxes" that are accessed by some set of generic APIs by higher-layer processes. OSes can vary widely in terms of what types of I/O APIs they provide to upper layers. For example, under Jbed, or any Java-based scheme, all resources (including I/O) are viewed and structured as objects. VxWorks, on the other hand, provides a communications mechanism, called *pipes*, for use with the vxWorks I/O subsystem. Under vxWorks, pipes are virtual I/O devices that include underlying message queue associated with that pipe. Via the pipe, I/O access is handled as either a stream of bytes (*block* access) or one byte at any given time (*character* access).

In some cases, I/O hardware may require the existence of OS buffers to manage data transmissions. Buffers can be necessary for I/O device management for a number of reasons. Mainly they are needed for the OS to be able to capture data transmitted via block access. The OS stores within buffers the stream of bytes being transmitted to and from an I/O device independent of whether one of its processes has initiated communication to the device. When performance is an issue, buffers are commonly stored in cache (when available), rather than in slower main memory.

9.5 OS Standards Example: POSIX (Portable Operating System Interface)

As introduced in Chapter 2, standards may greatly impact the design of a system component—and operating systems are no different. One of the key standards implemented in off-the-shelf embedded OSes today is portable operating system interface (POSIX). POSIX is based upon the IEEE (*1003.1-2001)* and The Open Group (*The Open Group Base Specifications Issue 6*) set of standards that define a standard operating system interface and environment. POSIX provides OS-related standard APIs and definitions for process management, memory management, and I/O management functionality (see Table 9-3).

Table 9-3: POSIX functionality [9-14]

OS Subsystem	Function	Definition
Process Management		
	Threads	Functionality to support multiple flows of control within a process. These flows of control are called threads and they share their address space and most of the resources and attributes defined in the operating system for the owner process. The specific functional areas included in threads support are: • Thread management: the creation, control, and termination of multiple flows of control that share a common address space. • Synchronization primitives optimized for tightly coupled operation of multiple control flows in a common, shared address space.
	Semaphores	A minimum synchronization primitive to serve as a basis for more complex synchronization mechanisms to be defined by the application program.
	Priority scheduling	A performance and determinism improvement facility to allow applications to determine the order in which threads that are ready to run are granted access to processor resources.

Table 9-3: POSIX functionality [9-14] (continued)

OS Subsystem	Function	Definition
Process Management	Real-time signal extension	A determinism improvement facility to enable asynchronous signal notifications to an application to be queued without impacting compatibility with the existing signal functions.
	Timers	A mechanism that can notify a thread when the time as measured by a particular clock has reached or passed a specified value, or when a specified amount of time has passed.
	IPC	A functionality enhancement to add a high-performance, deterministic interprocess communication facility for local communication.
Memory Management		
	Process memory locking	A performance improvement facility to bind application programs into the high-performance random access memory of a computer system. This avoids potential latencies introduced by the operating system in storing parts of a program that were not recently referenced on secondary memory devices.
	Memory mapped files	A facility to allow applications to access files as part of the address space.
	Shared memory objects	An object that represents memory that can be mapped concurrently into the address space of more than one process.
I/O Management	Synchronionized I/O	A determinism and robustness improvement mechanism to enhance the data input and output mechanisms, so that an application can ensure that the data being manipulated is physically present on secondary mass storage devices.
	Asynchronous I/O	A functionality enhancement to allow an application process to queue data input and output commands with asynchronous notification of completion.
…	…	…

How POSIX is translated into software is shown in Examples 9-16 and 9-17, examples in Linux and vxWorks of POSIX threads being created (note the identical interface to the POSIX thread create subroutine).

Example 9-16: Linux POSIX example [9-3]

Creating a Linux POSIX thread:

```
if(pthread_create(&threadId, NULL, DEC threadwork, NULL)) {
 printf("error");
 …
 }
```

Here, threadId is a parameter for receiving the thread ID. The second argument is a thread attribute argument that supports a number of scheduling options (in this case

NULL indicates the default settings will be used). The third argument is the subroutine to be executed upon creation of the thread. The fourth argument is a pointer passed to the subroutine (i.e., pointing to memory reserved for the thread, anything required by the newly created thread to do its work etc).

EXAMPLE 9-17: vxWorks POSIX example [9-4]

Creating a POSIX thread in vxWorks:

```
….
pthread_t tid;
int ret;

/* create the pthread with NULL attributes to designate default values */
ret = pthread_create(&threadId, NULL, entryFunction, entryArg);
….
```

Here, threadId is a parameter for receiving the thread ID. The second argument is a thread attribute argument that supports a number of scheduling options (in this case NULL indicates the default settings will be used). The third argument is the subroutine to be executed upon creation of the thread. The fourth argument is a pointer passed to the subroutine (i.e., pointing to memory reserved for the thread, anything required by the newly created thread to do its work, etc).

Essentially, the POSIX APIs allow for software that is written on one POSIX-compliant OS to be easily ported to another POSIX OS, since by definition the APIs for the various OS system calls must be identical and POSIX compliant. It is up to the individual OS vendors to determine how the internals of these functions are actually performed. This means that, given two different POSIX compliant OSes, both probably employ very different internal code for the same routines.

9.6 OS Performance Guidelines

The two subsystems of an OS that typically impact OS performance the most, and differentiate the performance of one OS from another, are the memory management scheme (specifically the process swapping model implemented) and the scheduler. The performance of one virtual memory-swapping algorithm over another can be compared by the number of page faults they produce, given the same set of memory references—that is, the same number of page frames assigned per process for the exact same process on both OSes. One algorithm can be further tested for performance by providing it with a variety of different memory references and noting the number of page faults for various number of page frames per process configurations.

While the goal of a scheduling algorithm is to select processes to execute in a scheme that maximizes overall performance, the challenge OS schedulers face is that there are a number of performance indicators. Furthermore, algorithms can have opposite effects on an indicator, even given the exact same processes. The main performance indicators for scheduling algorithms include:

- *Throughput*, which is the number of processes being executed by the CPU at any given time. At the OS scheduling level, an algorithm that allows for a significant number of larger processes to be executed before smaller processes runs the risk of having a lower throughput. In a SPN (shortest process next) scheme, the throughput may even vary on the same system depending on the size of processes being executed at the moment.
- *Execution time*, the average time it takes for a running process to execute (from start to finish). Here, the size of the process affects this indicator. However, at the scheduling level, an algorithm that allows for a process to be continually preempted allows for significantly longer execution times. In this case, given the same process, a comparison of a non-preemptable vs. preemptable scheduler could result in two very different execution times.
- *Wait time*, the total amount of time a process must wait to run. Again this depends on whether the scheduling algorithm allows for larger processes to be executed before slower processes. Given a significant number of larger processes executed (for whatever reason), any subsequent processes would have higher wait times. This indicator is also dependent on what criteria determines which process is selected to run in the first place—a process in one scheme may have a lower or higher wait time than if it is placed in a different scheduling scheme.

On a final note, while scheduling and memory management are the leading components impacting performance, to get a more accurate analysis of OS performance one must measure the impact of both types of algorithms in an OS, as well as factor in an OS's *response time* (essentially the time from when a user process makes the system call to when the OS starts processing the request). While no one factor alone determines how well an OS performs, OS performance in general can be *implicitly* estimated by how hardware resources in the system (the CPU, memory, and I/O devices) are utilized for the variety of processes. Given the right processes, the more time a resource spends executing code as opposed to sitting idle *can be* indicative of a more efficient OS.

9.7 OSes and Board Support Packages (BSPs)

The ***board support package*** (BSP) is an optional component provided by the OS provider, the main purpose of which is simply to provide an abstraction layer between the operating system and generic device drivers.

A BSP allows for an OS to be more easily ported to a new hardware environment, because it acts as an integration point in the system of hardware dependent and hardware independent source code. A BSP provides subroutines to upper layers of software that can customize the hardware, and provide flexibility at compile time. Because these routines point to separately compiled device driver code from the rest of the system application software, BSPs provide run-time portability of generic device driver code. As shown in Figure 9-39, a BSP provides architecture-specific device driver configuration management, and an API for the OS (or

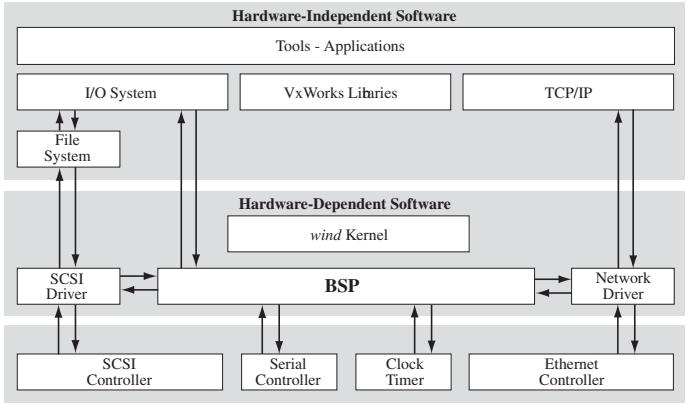


Figure 9-39: BSP within Embedded Systems Model [9-4]

higher layers of software) to access generic device drivers. A BSP is also responsible for managing the initialization of the device driver (hardware) and OS in the system.

The device configuration management portion of a BSP involves architecture-specific device driver features, such as constraints of a processor's available addressing modes, endianess, and interrupts (connecting ISRs to interrupt vector table, disabling/enabling, control registers) and so on, and is designed to provide the most flexibility in porting generic device drivers to a new architecture-based board, with its differing endianess, interrupt scheme, and other architecture-specific features.

9.8 Summary

This chapter introduced the different types of embedded OSes, as well as the major components that make up most embedded OSes. This included discussions of process management, memory management, and I/O system management. This chapter also discussed the POSIX standard and its impact on the embedded OS market in terms of what function requirements are specified. The impact of OSes on system performance was discussed, as well as OSes that supply a board-independent software abstraction layer, called a board support package (BSP).

The next chapter, Chapter 10, is the last of the software chapters and discusses middleware and application software in terms of their impact on an embedded architecture.

Chapter 9 Problems

1. [a] What is an operating system (OS) ?
   [b] What does an operating system do?
   [c] Draw a diagram showing where the operating system fits in the Embedded Systems Model.

2. [a] What is a kernel?
   [b] Name and describe at least two functions of a kernel.

3. OSes typically fall under one of three models:
   A. monolithic, layered, or microkernel.
   B. monolithic, layered, or monolithic-modularized.
   C. layered, client/server, or microkernel.
   D. monolithic-modularized, client/server, or microkernel.
   E. None of the above.

4. [a] Match the type of OS model to Figures 9-40a, b, and c.
   [b] Name a real-world OS that falls under each model.

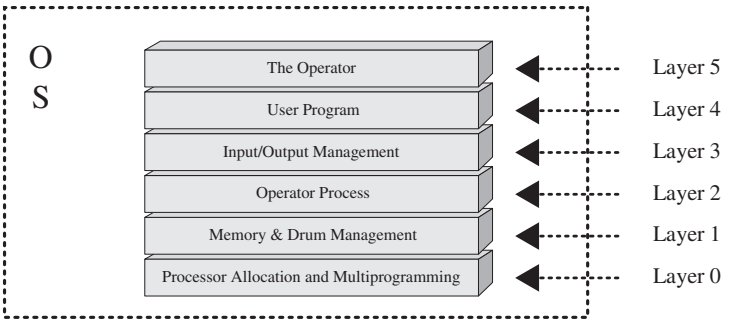


Figure 9-40a: OS block diagram 1

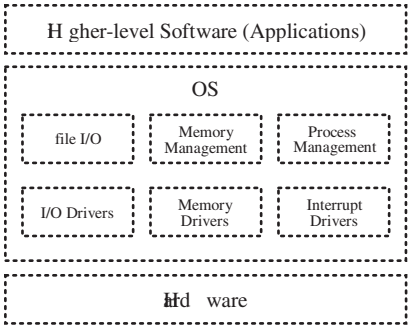


Figure 9-40b: OS block diagram 2

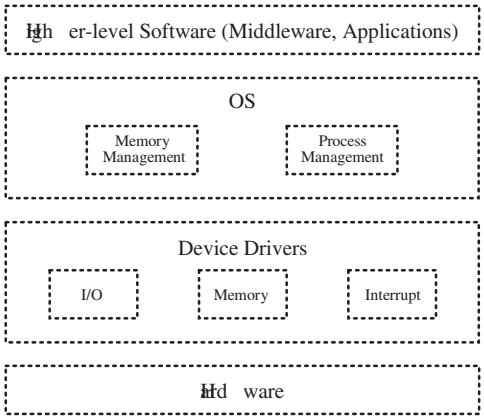


Figure 9-40c: OS block diagram 3

5. [a] What is the difference between a process and a thread?
   [b] What is the difference between a process and a task?

6. [a] What are the most common schemes used to create tasks?
   [b] Give one example of an OS that uses each of the schemes.

7. [a] In general terms, what states can a task be in?
   [b] Give one example of an OS and its available states, including the state diagrams.

8. [a] What is the difference between preemptive and non-preemptive scheduling?
   [b] Give examples of OSes that implement preemptive and non-preemptive scheduling.

9. [a] What is a real time operating system (RTOS)?
   [b] Give two examples of RTOSes.

10. [T/F] A RTOS does not contain a preemptive scheduler.

11. Name and describe the most common OS intertask communication and synchronization mechanisms.

12. [a] What are race conditions?
    [b] What are some techniques for resolving race conditions?

13. The OS inter-task communication mechanism typically used for interrupt handling is:
    A. a message queue.
    B. a signal.
    C. a semaphore.
    D. All of the above.
    E. None of the above.

14. [a] What is the difference between processes running in kernel mode and those running in user mode?
    [b] Give an example of the type of code that would run in each mode.

15. [a] What is segmentation?
    [b] What are segment addresses made up of?
    [c] What type of information can be found in a segment?

16. [T/F] A stack is a segment of memory that is structured as a FIFO queue.

17. [a] What is paging?
    [b] Name and describe four OS algorithms that can be implemented to swap pages in and out of memory.

18. [a] What is virtual memory?
    [b] Why use virtual memory?

19. [a] Why is POSIX a standard implemented in some OSes?
    [b] List and define four OS APIs defined by POSIX.
    [c] Give examples of three real-world embedded OSes that are POSIX compliant.

20. [a] What are the two subsystems of an OS that most impact OS performance?
    [b] How do the differences in each impact performance?

21. [a] What is a BSP?
    [b] What type of elements are located within a BSP?
    [c] Give two examples of real-world embedded OSes that include a BSP.