11 Embedded Operating systems & Multitasking

11.1 Embedded Operating Systems

Some embedded systems have to deal with large amounts of data. They have to receive input data and they also need to output their results. In the beginning, users wrote their own computer programs to do this. Increasingly, many of these programs had similar features. After all, most of the devices operated the same way.

Also, with increasing computing power, more data was collected to be processed. Users had to organize their data on the hard disk. They also had to be able to transfer data between one another. They would also want automated ways of letting their programs run. Let's look at these tasks:

Getting input from a user
Outputting to a display and/or a printer
Creating, reading and writing to files
Load, run and terminate a computer program

Over the years, programs to do these tasks were collected and standardized so they could work with each other. Such a collection of programs was called an Operating System (OS).
Common examples are Linux and the Windows family.

Smaller embedded systems may not need all the features of an operating system. In fact, these features, if included without thought, will increase the size of the embedded system needlessly and may even prevent the system from running! That is why there is a wide variety of embedded operating systems.

Let us look at the development of some of these concepts:

User Interface:
From text-only outputs of early computers, we now have GUI (Graphical User Interfaces).

File System:
This allows for the orderly management of data. Facilities may range from a simple 'copy' or 'move' to automatic recovery of data when a power failure occurs.

Task Management: Loading/Running/Terminating a program
Users can schedule a computer to load various programs to perform the tasks required of it. The computer has to keep track of every process that it is executing. It needs to reserve memory to run, hard disk space and to reclaim these resources when done.

Note that a computer can only execute one program at a time. But because it executes instructions at a much higher rate than humans, it can execute several programs a portion at a time, it can give the impression that it is doing several things at the same time. For example, a computer can do printing while allowing a user to enter data into a file, at the same time. This

feature is called multitasking and with the increasing power of embedded processors, is frequently being used.

An *embedded operating system* is an operating system operating for embedded

computer systems. These operating systems are designed to be very compact and efficient, forsaking many functions that non-embedded computer operating systems provide, and which may not be used by the specialized applications they run. They are frequently also *real-time operating systems*, requiring a fixed time to respond to I/O requests, Multitasking is one the main features of a real-time operating system.

11.2 Multitasking Systems

There are various definitions of the terms task, process and thread. In this chapter, we shall use a descriptive approach. A program as it exists on the disk contains both code and data. When loaded into memory by the OS, the code becomes a set of instructions in memory which is called a task. Tasks that are *separate* executing programs are called processes while tasks that are executed in the context of a *single* program are called threads. When we use the word tasks here, we are referring to both processes and threads. In embedded application, multi-threading is more commonly being used than multi-processing.

Multitasking is a method by which multiple tasks, also known as **process** and **threads** share common processing resources such as a CPU. In the case of a computer with a single CPU, only one task is said to be running at any point in time, meaning that the CPU is actively executing instructions for that task. Multitasking solves the problem by **scheduling** which task may be the one running at any given time, and when another waiting task gets a turn. The act of reassigning a CPU from one task to another one is called **a context switch**.
When context switches occur frequently enough the illusion of parallelism is achieved.

Using multitasking operating system has two important benefits. First, the embedded system can perform more work in the same amount of time. This is due to the fact that operating system can avoid the unnecessary delays encountered in executing a task. For example, the OS has to run two tasks, Task A and Task B. Task A runs to a point in its execution where it must stop and wait for input (perhaps waiting for data to arrive from network port). In a single-tasking system, the system waits to complete Task A before going to Task B. In a multitasking system, there is no need to wait for Task A to complete its task. The OS can run Task B while Task A is waiting for input. When Task A received its input, the OS stops running Task B and returns to Task A. The second main benefit is the system has the capability to handle multiple tasks “simultaneously”.

11.3 Multiprocessing and multithreading

A program consists of algorithms expressed in the program code. A process is the activity of performing work according to these algorithms. As shown in Figure 11.1, a process consists of a collection of memory areas allocated to the process by the OS, plus the current CPU state. When a program is launched, the OS creates a process and allocates a collection of memory areas (code, stack, data, heap and system memory).

A process owns its memory area. A running process is also defined by the current

CPU state. The CPU state consists of the general processor registers (IP, SP, Flags and other CPU registers) that can be modified by the application program. When a process is not running, its CPU state is saved in the process's system memory area.

Running more than one process at the same time is called multiprocessing. An OS runs multiple processes by constantly switching from one process's context to another (context switching). A process context consists of its CPU state and memory areas. The OS switches context by suspending the execution of the current process, saving its CPU state into its system memory area, and loading the context of the next process to run. A scheduler program in the OS determines when to terminate the current process and what is the next process to execute in a multiple process environment. In a real-time embedded system, each task must be executed in a predictable manner and within the time constraints. The scheduler program must be able to meet these requirements to support a real-time operating system.

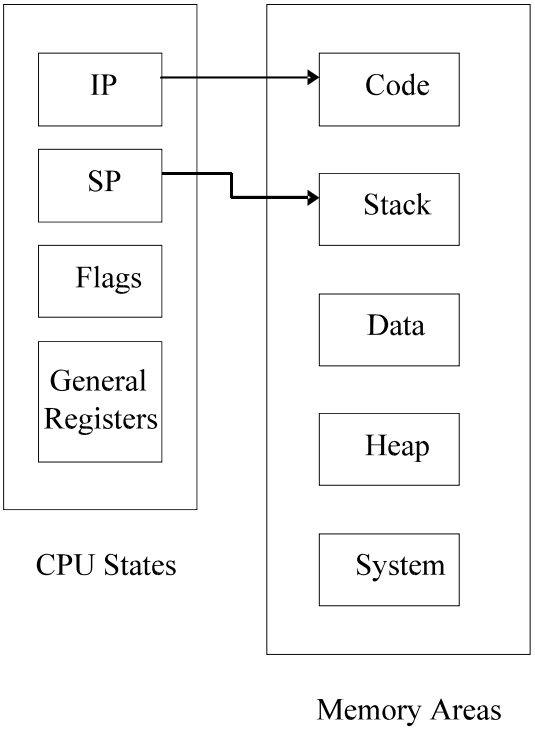


Fig 11.1 Context of a process

A thread is an independent flow of execution in a process. In a multithreading OS, a process consists of one or more threads. All threads in a process share the code, data, heap and system memory areas. As shown in Figure 11.2, each thread has a separate CPU state and a separate stack, a block of memory allocated out of the process' stack memory area. As all threads of a process share the same data and heap memory areas, all global variables in the process can be accessed by any of the threads. However, each thread has its own stack, all local variables and function arguments are private to the specific thread.

Because threads shared the same code and global data, they are tied much more closely than processes, and they tend to interact much more than separate processes. For this reason, *synchronization* objects are likely to be used more frequently in multithreading applications than multiprocessing applications.

Context switching between threads in the same process involves simply saving the CPU state for the current thread and loading the CPU state of the new thread. Because less

work is required for context switching between threads than between processes, threads are sometimes called lightweight processes.

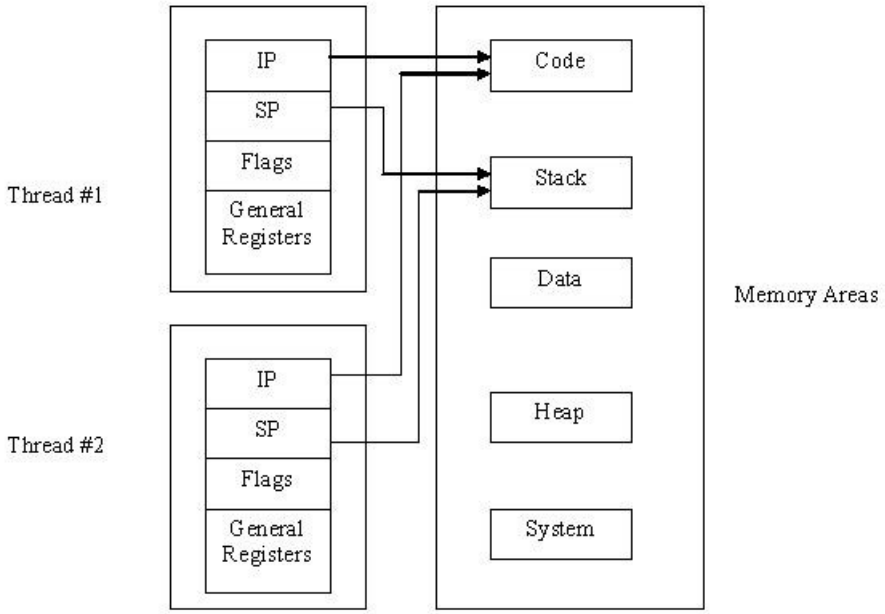


Fig 11.2 Context of a thread

Processes are normally created when programs are launched, either by user of the system or by another process calling the OS. In some systems, (such as Linux) processes can also *clone* themselves to create a new process, for example the fork() function calls. Processes created in this way often share the same code memory to conserve memory use, but they each get their own stack, data, heap and system data areas. Processes created by cloning may sound like threads, because they are running from the same program code. However, they are a process because they each have their own separate global data, unlike threads.

In summary, we see that threads are more desirable than tasks, as they use up less resource. However, threads have to be initiated by the programmer, whereas tasks are managed by the operating system as soon as a program is loaded into memory.

11.4 Scheduling

A multitasking OS worked by running a task and then switched to another task. This can be done in two ways:

i) cooperative - programs give way to one another when doing input/output for example, waiting for a user to press a key. This is very slow, from a computer point of view.
ii) Pre-emptive - programs forced to give way to one another, to prevent "hogging".

The amount of time an OS let a task execute is called a time slice, is often of a fixed duration. The OS partitions out a time slice for each task, one after another. When the time slice for

running a task ends, a program in the OS called the scheduler determines which task will have the next time slice.

A task can be one of the three states:

Running. In this state, the task is executing

Ready. In this state, the task is waiting for their turn for the CPU
Blocked. In this state, the task is waiting for something to happen.

A scheduler maintains one or more internal lists for keeping track of the state of each task. Typically, it has one ready list and a separate blocked list for each *synchronization object* on which tasks are waiting. The task at the head of the ready list is the next task to run. Tasks on any of the block lists are suspended. They are waiting for some events. Whenever an event occurs for which a task on the blocked list is waiting, the task is removed from the blocked list and placed onto the ready list, where it waits for its turn for execution.

How does a scheduler determine which task to run next? The answer depends on the scheduling algorithm used. Many type of scheduling algorithm are available, such as first in first out (FIFO) and round robin. For a real time embedded system, it is critical that the scheduling algorithm be deterministic, i.e. it is always possible to predict which task will run next. We will present the round robin scheduling here because it is simple and predictable.

Figure 11.3 shows how the round robin scheduling works. The contexts for the tasks that are ready to run are kept on a ready list, which is implemented as a linked list of task control blocks (TCB). Assuming all tasks has equal priority; the scheduler removes the TCB from the head of the ready list and makes its associated task the current executing task. This is referred to as switching-in the task's context. The task runs for a time slice, at the end of which the scheduler (awakened by the timer interrupt) saves the state of the task (referred to as switching-out the task's context), place its TCB at the end of list, pull the next TCB off the head of the list, and run the task. The scheduler continues to repeat the process, with each task running for an equal time slice, by proceeding through the list.

We have assumed the tasks have equal priority, but more often than not, some tasks are more important than others. The programmer can assign each task in the system a priority, a numeric value that assigns a level of importance to the task. When it's time for the scheduler to choose the next task to run, it selects the task from the ready list that has the highest priority. Implementing task priorities are important in real time scheduling algorithm. A real time OS has to implement deterministic scheduling, so the programmer can, given any circumstances, always identify which task in the application will run. The scheduler in a real time OS always selects the highest priority task on the ready list. It is immediately scheduled again and get the next time slice. By contrast, the Windows OS occasionally boosts the priorities of low priority tasks to ensure they get some CPU time. To guarantee that the system meets its real time deadlines, the programmer has to have complete control over which task or tasks are eligible to run.

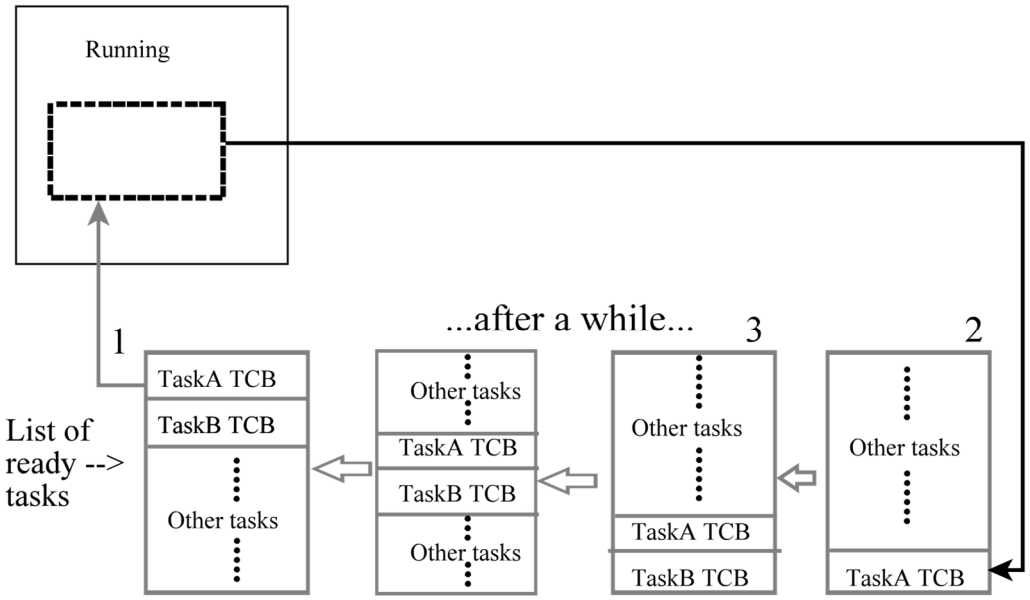


Fig 11.3 Round robin scheduling algorithm

11.5 Synchronization

In a multitasking system, tasks can interact. Sometimes they interact directly with each other; other times they interact through shared resources. These interactions must be coordinated, or synchronized, to prevent what is called a race condition. A race condition occurs when the outcome of the computation of two or more tasks depends on how quickly the tasks execute.

Let's have a look at what happens to make a race condition. Suppose both Task 1 and Task 2 access a global counter variable. Each task increments the counter variable using the following code:

```
Counter++;
```

After compilation, the generated processor might look like this (using pseudocode)"

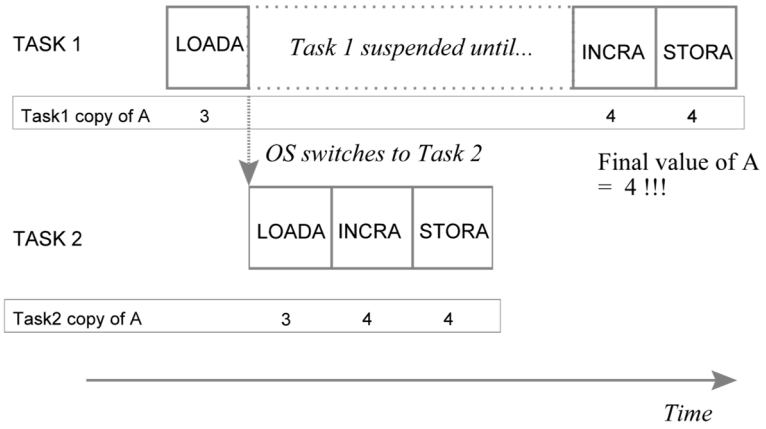
```
LOADA     counter
INCRA
STOREA    counter
```

The first instruction loads the value of counter into the CPU's accumulator. The next instruction adds 1. The final instruction stores the result back to counter.

Suppose Task 1 reaches the series of instructions when the counter has a value of 3. It executes LOADA instruction and load 3 into the accumulator. Before Task 1 can execute the INCRA instruction, however, the task's time slice ends and the scheduler switches it out. The content of the accumulator (a value of 3) is stored with the task context when it is switched out.

Task 1 resumes

The scheduler starts Task 2. Task 2 reaches these instructions and executes LOADA. Because Task 1 did not store the result in counter, Task 2 fetches 3 into the accumulator also. Task 2 then executes the INCRA and STOREA instructions and continues with whatever instructions follow. The content of global variable counter is now 4.



At some time later, Task 1 is rescheduled. The scheduler reloads Task 1's context and places 3 back to the accumulator. You can now see the problem. Task 1 restarts, executes INCRA and STOREA instructions and proceed with whatever instructions follow. The content of the global variable counter is again 4. Had Task 1 not been preempted at the moment it was, it would have executed the INCRA and STOREA instructions, and the global counter would be 5. Because neither task guarded its access to the counter global variable, a race condition occurred.

The code section that accesses the shared resource is called a critical section. In the example above, the critical section is just one C statement. To avoid race conditions, we need tasks to be *mutually exclusive*, to ensuring that only one task can be executing in a critical section at any moment.

Synchronization objects are used to synchronize the use of shared resource and prevent race conditions. We will look at three types of synchronization object,

Mutex objects
Semaphore objects
Event objects

11.5.1 Mutex Objects

A mutex object, sometimes called a critical section object, derives its name from its use in coordinating mutually exclusive accesses to a shared resource. A mutex object can be in one of the two states, owned or free. The mutex object can be owned by only one task at a time. Operating systems that support mutex provide two function calls for manipulating them: a wait call and a release call.

Operating systems that support mutexes provide two function calls for manipulating them, a wait call and a release call. When a task wants to acquire a mutex, it issues a wait call. If the mutex is free, the wait call returns immediately and the calling task has acquired ownership of the mutex. If the mutex is already owned, the wait call doesn't return, and the calling task is blocked. A task waiting on the mutex becomes unblocked when the owner of the mutex issues a release call on the mutex.

A mutex is similar to a room built around a critical section of code. The room has one door in and one door out. Furthermore, a guard at the door admits only one person (task) into the room (critical section) at any given time. Tasks arriving at the entrance while the room is

occupied must wait at the door until the room becomes empty. Also, the guard (operating system) sees to it that those waiting at the door are admitted in task priority order.

11.5.2 Semaphore Objects

A semaphore consists of a data item, which we will call a count, and a pair of operations, *wait* and *release*, which are implemented as OS APIs. A task requesting access to whatever resource the semaphore guards need to perform a *wait* operation on the semaphore. When a task is finished with the resource, it performs a *release* on the semaphore.

The semaphore's count determines what happens when a wait or release operation executes. When the program creates a semaphore, it sets the semaphore's count to an initial positive value. The value indicates the number of tasks the semaphore will let pass before closing the door. Whenever a task performs a wait operation on the semaphore, the OS first checks to see whether the semaphore's count is greater than 0. If so, the call operation succeeds (task is allowed to continue) and the semaphore's count is decrease by one. If the count is 0, the task is blocked.

When a task has finished using whatever resource the semaphore is guarding, the task issues a release call on the semaphore. The release call causes the OS to increment the semaphore's count by 1. Semaphore objects allow multiple tasks to simultaneously access a common resource, such as network connection.

11.5.3 Event Objects

Event objects are typically used for synchronizing tasks processing rather than for controlling access to shared resources. It is a mechanism that lets a task go to sleep until, for example, some data is ready for it to process or a request is ready for it to service.

Operating system functions allow tasks to set to **signaled** as well as reset to **nonsignaled** an event object. Event objects can be created in either the signaled or the nonsignaled state. When a task waits on a non-signaled event object, the task is blocked until another thread sets the event object. When a task waits on a signaled event object, the task does not block. A common use of an event object is to notify a task when a pending I/O operation has been completed.

There are two types of event objects, i.e. auto reset events and manual reset events. Auto reset events are automatically reset to non-signaled state when a single waiting task is released (unblock). If multiple tasks are waiting, only the highest-priority waiting task is released. The next time a running task set the event, another waiting task is released, and so on. Manual reset events remain set to the signaled state after a task sets them. If multiple tasks are waiting on the event, they are all released (unblocked) when another task set the event. Subsequent tasks that wait on the even object proceed immediately, without blocking. To cause task to block on the event again, a task must explicitly reset the event to its nonsignalled state. An example of setting up multiple threads is shown in the appendix.

Appendix

Creating multithreading application

When a process begins, it contains a single thread. Additional threads within the process must be started explicitly. A thread can be started by a call to *pthread_create()*,

The *pthread_create()* function's prototype is

```
int pthread_create(pthread_t *thread, const phread_attr_t *attr, void *(*start_routine)(void *), void *arg)
```

Parameters

thread - Once the function is called successfully, the variable whose address is passed as first argument will hold the thread ID of the newly created thread.

attr - contain certain attributes which we want the new thread to contain. It could be priority, stack address, stack size etc. Set to NULL if default thread attributes are used.

*void * (*start_routine)* - pointer to the function to be threaded. Function has a single argument: pointer to void. Each thread starts with a function and that functions address is passed here as the third argument so that the kernel knows which function to start the thread from.

arg - As the function (whose address is passed in the third argument above) may accept some arguments, we can pass these arguments in form of a pointer to a void type. A void type was chosen because if a function accepts more than one argument then this pointer could be a pointer to a structure that may contain these arguments.

Initializing a Mutex

The *pthread_mutex_init()* function initialises the mutex referenced by *mutex* with attributes specified by *attr*. If *attr* is NULL, the default mutex attributes are used; the effect is the same as passing the address of a default mutex attributes object. Upon successful initialisation, the state of the mutex becomes initialised and unlocked.

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t *attr)
```

Parameters

mutex - the address of the variable to contain a mutex object.

attr - the address of the variable containing the mutex attributes object..

Locking and unlocking a Mutex

After initilizing a mutex, any critical region in the code can be locked using the *pthread_mutex_lock()* function.

The *pthread_mutex_lock()* function's prototype is

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
```

Parameters

mutex - the address of the mutex to lock

If the mutex is already locked by another thread, the thread waits for the mutex to become available. The thread that has locked a mutex becomes its current owner and remains the owner until the same thread has unlocked it.

To unlock and release the mutex, the function to call is *pthread_mutex_unlock()*.

```
int pthread_mutex_unlock(pthread_mutex_t *mutex)
```

Waiting for thread termination

pthread_join() is used when one thread waits for another one to finish. The two threads work in parallel, then they must combine the results they obtained. One of them calls join and waits for the other one to exit, so it can collect its result.

```
int pthread_join(pthread_t thread, void **status)
```

Parameters

thread - the thread to wait for

status - the location where the exit status of the joined thread is stored. This can be set to NULL if the exit status is not required.

Wait on a condition

To let a thread sleep, condition variable can be used. In C under Linux, there is a function pthread_cond_wait() to wait or sleep.
On the other hand, there is a function pthread_cond_signal() to wake up sleeping or waiting thread.
Threads can wait on a condition variable.

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex)
```

Parameters

cond – pointer to the condition variable.

mutex - Pointer to the mutex associated with the condition variable *cond*