Introduction

Synchronization provides a method for guaranteeing consistent data and coordination of threads of code. We will concentrate on process and thread oriented synchronization considerations. Students will also understand methods of determining completion of Asynchronous System Services using event flags and Asynchronous System Traps.

Objectives

Students should be able to:

- Describe the difference between synchronizing with the system and other processes.
- Identify the number, types and use of event clusters.
- Describe the control flow associated with AST delivery.
- Use timers and time related services.

Topics

- Synchronization
- Local Event Flags
- Asynchronous System Traps (ASTs)
- Timers and Time

Synchronization

In any interrupt-driven, multi-user system, synchronization is crucial to guarantee consistent data and coordinate process activities.

- Three general forms of synchronization must be considered when designing system applications for OpenVMS:

1. System synchronization for determining I/O completion, timer expiration, getting information on other processes, etc.
2. Inter-process synchronization to coordinate access to shared data, such as files and shared memory (global sections).
3. Multi-threaded/AST driven code to coordinate access to shared data in process virtual address space.

- System synchronization is performed through:
  - Event flags (Normally, local)
  - Asynchronous System Traps (ASTs)
  - The "I/O" status block
- Inter-process synchronization is performed through:
  - Common event flags
  - Locks
  - Special instructions (load lock/store conditional, interlocked queue instructions (PAL), and memory barriers(PAL))
- Multi-threaded/AST driven synchronization is performed through:
  - Special instructions (load lock/store conditional, interlocked queue instructions (PAL), and memory barriers(PAL))
  - Parallel Processing Library (PPL$) semaphores and *memory barriers*
  - DECThreads mutexes

Note

Other forms of synchronization are available for kernel mode code.

Event Flags

OpenVMS provides event flags and associated System Services to synchronize with a system event.

- Four event flag clusters with 32 event flags each are provided for process use.
- Two of the clusters are considered local and are available automatically. Two are common and must be associated with the process, prior to use.

Event flag clusters

Cluster	Cluster	Flags	Considerations
0	local	0 through 31	Flags 24 through 31 are reserved for system usage
1	local	32 through 63	May be allotted using LIBGET_EF, LIBFREE_EF and LIB$RESERVE_EF
2	common	64 through 95	Must be associated
3	common	96 through 127	Must be associated
4	local	128	pseudo-cluster, used for EFN$C_ENF (see SYS$SYNCH)

- Local Event Flag services

Note

Most asynchronous System Services offer a *W* form which will implicitly wait, e.g. **sys$qio** and **sys$qiow**.

Wait for event flag

`sys$waitfr(efn)`

Wait for multiple flags

`sys$wflor(efn,mask)`

`sys$wfland(efn,mask)`

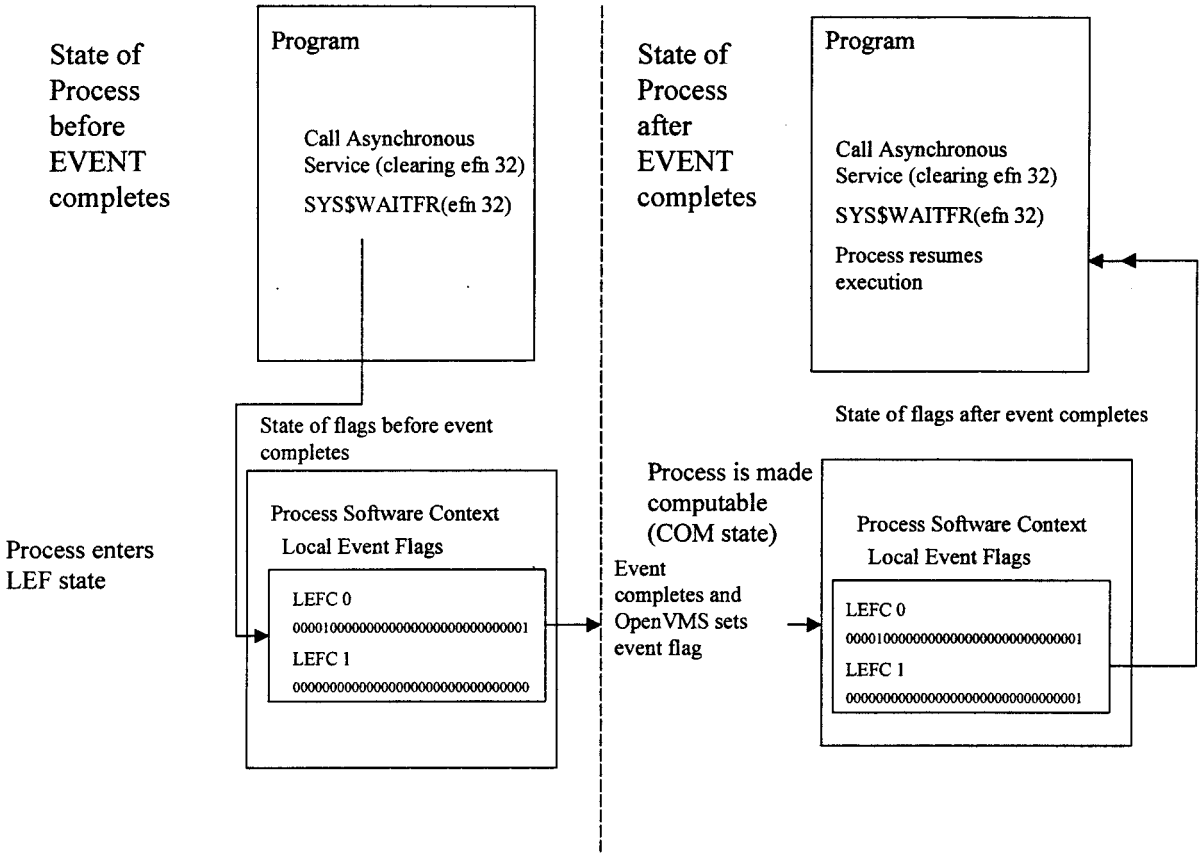
Read event flag cluster

`sys$readef(efn,state)`

Set event flag

`sys$setef(efn)`

Flow Using Event Flags for Synchronization



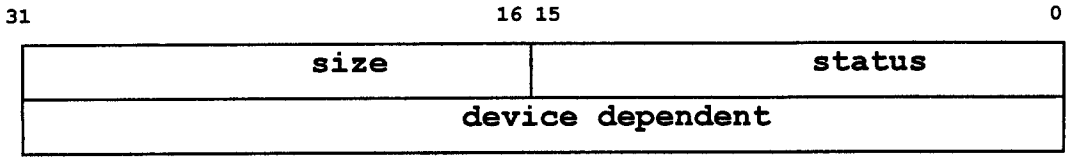
General flow when using event flags for synchronization

The "I/O" Status Block

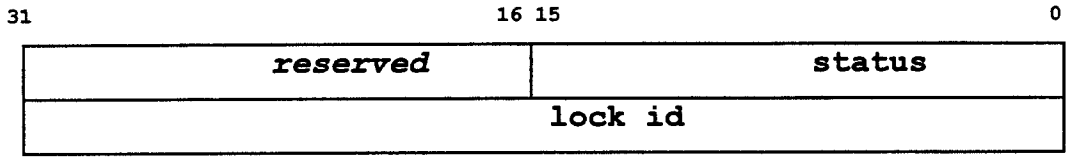
The return condition code from a System Service call identifies whether the system service was called properly. A successful condition code does not guarantee that an asynchronous service, such as **sys$qio** or **sys$enq**, actually completed properly.

- Asynchronous system services support an *asynchronous completion status block* to check for successful completion of a request. This block is commonly referred to as an I/O status block (iosb), although lock management calls it a lock status block (lksb).

Format of iosb



Format of lksb



- The iosb/lksb is initially set to 0 to indicate that the request has not completed.

SYS$SYNCH System Service

Besides checking the status field directly for successful completion, you can also use the iosb with the **sys$synch** System Service. This may be useful if there is a chance of having the same event flag set multiple times.

- Format of **sys$synch**:

  `sys$synch(efn,iosb)`

- The **iosb** arguments should point to unique I/O status blocks for unique events.

The flag **EFN$C_ENF** indicates that you are waiting for an event with no specific associated event flag.

- This flag should, generally, only be used with the SYS$SYNCH system service.
- You are effectively waiting on the iosb, not the event flag being set.
- Whenever an event flag is posted, with a flag of EFN$C_ENF (128), all kernel threads are activated. If the event was not associated with this thread the iosb will still be clear, and the process will reenter LEF wait state.
- This minimizes potential conflicts with different threads, accidently, reusing event flags used by other threads.

Time and Time-Related Services

OpenVMS time string representations

- Absolute time is represented as: **DD-MMM-YYYY:hh:mm:ss.cc**
- Delta time is represented as: **DDDD-hh:mm:ss.cc**

Internal time representations

- Absolute time is passed as a positive quadword. It is maintained as the elapsed 100ns intervals since 17-NOV1858:00:00:00.00.
- Delta time is passed as a negated quadword. It is the number of 100ns from now.

Waiting for time-based events (granularity ~10ms)

```
sys$setimr(efn,daytim,astadr,reqidt,flags)
```

`if flags == 1` timer is cpu time based, else elapsed time.

```
sys$schdwk(pidadr,prcnam,daytim,reptim)
```

To cancel a timer/wake up

```
sys$cantim(reqidt,accmode)
sys$canwak(pidadr,prcnam)
```

Cancels all wakeups.

Convert time from internal binary to ascii

```
sys$asctim(timlen,timbuf,timadr,cvtflg)
```

Timer Example

```
$ type timer.c
```

```
/* Sample of using $setimr system service.
 */
#include <stdio.h>
#include <ssdef.h>
#include <starlet.h>
#include <lib$routines.h>
#include <descrip.h>
const       int     three_sec[2] = {-3*1000*1000*10,-1};
#define MAX_TIME_STR 23
#define     TIME_EF 32
#define sys_stat(STS) if(!((STS)&1)) sys$exit(STS)
int  main(void)
{
     int     status;
     int     i;
     char    time_str[MAX_TIME_STR];
     struct dsc$descriptor_s cur_time =
                          {sizeof(time_str),0,0};

/* Set up descriptor to hold time. */
     cur_time.dsc$a_pointer = time_str;
/* Display time. */
     printf("Time:\n");
     status = sys$asctim(&cur_time.dsc$w_length,&cur_time);
     sys_stat(status);
     lib$put_output(&cur_time);
/* Wait 3 seconds. */
     status = sys$setimr(TIME_EF,three_sec,0,0,0);
sys_stat(status);
     status = sys$waitfr(TIME_EF);
     sys_stat(status);
/* Redisplay time. */
     cur_time.dsc$w_length = sizeof(time_str);
     status = sys$asctim(&cur_time.dsc$w_length,&cur_time);
     sys_stat(status);
     printf("Time after waiting 3 seconds:\n");
     lib$put_output(&cur_time);
     return(SS$_NORMAL);
}
```

```
$ run timer
```

```
Time:
22-JUL-1996 21:28:28.66
Time after waiting 3 seconds:
22-JUL-1996 21:28:31.66
$
```

Additional Time and Time-Related Services

The following services are also available for time conversion:

Convert time from ascii to internal binary

`sys$bintim(timbuf,timadr)`

Get current time

`sys$gettim(timadr)`

Convert time to sortable numeric representation

`sys$numtim(timbuf,timadr)`

Note
timbuf is a 7 word structure

Set time

`sys$setime(timadr)`

Time must be absolute, need CMKRNL privilege.

For Universal Coordinated Time (UTC) use the equivalent

`sys$timcon, sys$ascutc, sys$binutc, sys$getutc, sys$numutc`

- Many run-time library routines support time conversion and timers, including:

`lib$add_times`	`lib$convert_date_string`
`lib$cvtf_from_internal_time`	`lib$cvtf_to_internal_time`
`lib$cvt_from_internal_time`	`lib$cvt_to_internal_time`
`lib$date_time`	`lib$day`
`lib$day_of_week`	`lib$format_date_time`
`lib$free_date_time_context`	`lib$get_date_time_format`
`lib$init_date_time_context`	`lib$init_timer`
`lib$multf_delta_time`	`lib$mult_delta_time`
`lib$show_timer`	`lib$sys_asctim`

Asynchronous System Traps

An Asynchronous System Trap (AST) routine is a procedure which is called upon completion of an asynchronous event. It allows for notification of event completion, when the process does not wait for the event to complete.

- AST routine addresses are passed to System Services through the **astadr** argument. An optional argument may be passed to the AST routine by specifying the **astprm** argument on the System Service call.
- ASTs are non-preemptive within the same access mode in the same *kernel thread.* However, on Alpha systems with multithreading support, ASTs may run concurrently and order of delivery of the ASTs is not guaranteed.
- To synchronize access to shared data between the main program and AST routines, the main program may call the **sys$setast** system service to block AST delivery. Note, however, that an AST may already be running in another kernel thread
- The AST routine actually receives 5 arguments:

```
astrpm
saved R0
saved R1
saved PC
saved PS
```

- You may test AST level code using `sys$dclast`.

AST Flow

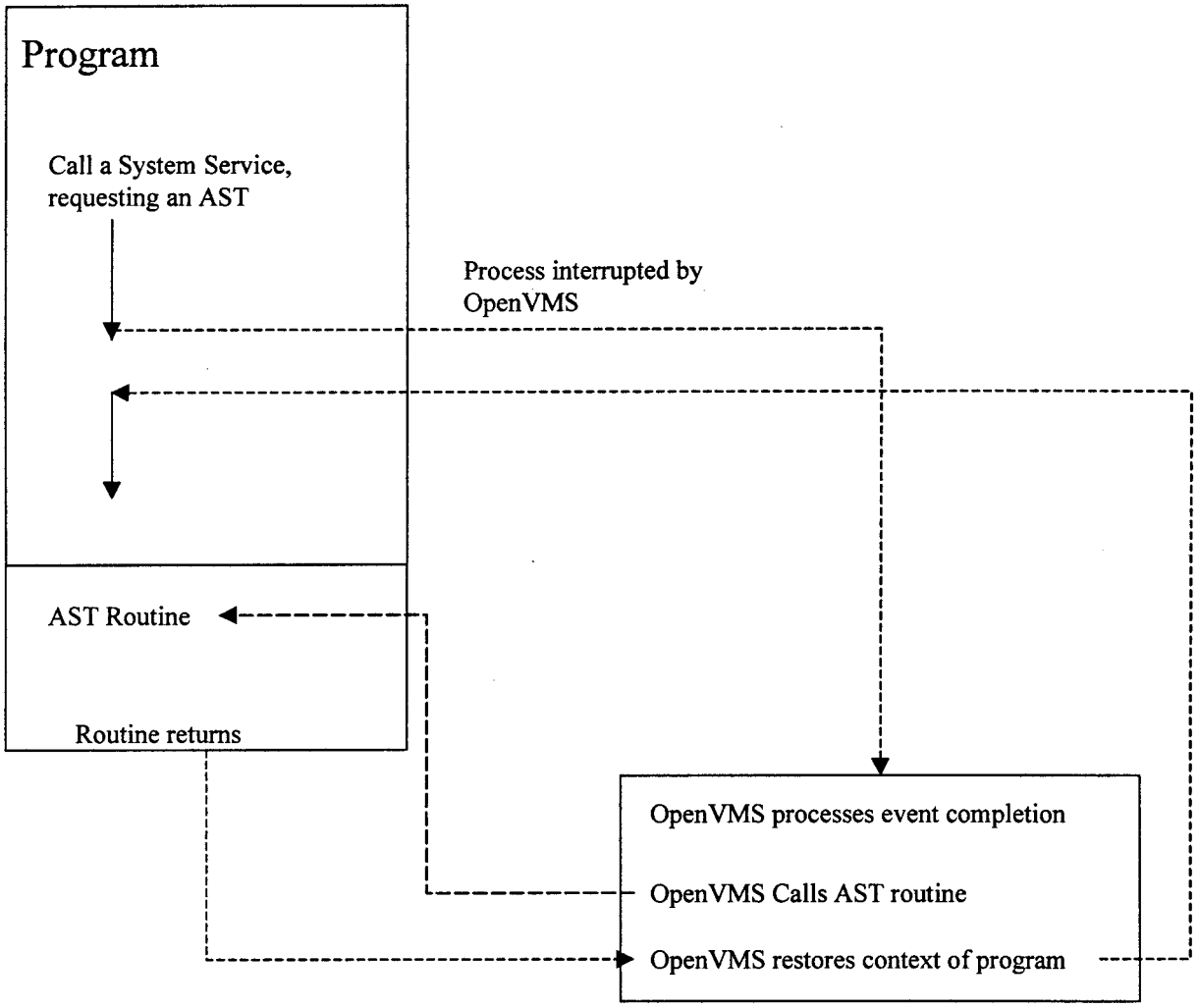


Illustration of control flow when an AST routine is associated with event completion.

Asynchronous System Traps Example

```
$ type ast_run.c
/* Example of using ASTs with the setimr system service.

*/
#include        <stdio.h>
#include        <ssdef.h>
#include        <starlet.h>
#include        <lib$routines.h>
#include        <descrip.h>
#define sys_stat(STS) if(!((STS)&1)) sys$exit(status);
const   int     ten_ms[2] = {-10*1000*10,-1};
#define E_EF 32
#define C_EF 33
#define EFC_BASE_FLAG 32
#define DISP_ITERATIONS 1000000
#define ITERATIONS DISP_ITERATIONS*10
#define ELAPSED_TIMER 0
#define CPU_TIMER 1
#define TICK_TIMER 1
#define ID_ARG 0
#define CTR 1
#define EF 2
#define AST_ARGS 3

int     main(void)
{
        int     ast_rtn(int *reqid);
        int     i;
        int     status;
/* Timer id argument, has id and counter */
        int     rid1[AST_ARGS] = {ELAPSED_TIMER,0,E_EF};
        int     rid2[AST_ARGS] = {CPU_TIMER,0,C_EF};

/* Set up an elapsed time timer with AST. */
        status = sys$setimr(E_EF,ten_ms,ast_rtn,rid1,0);
        sys_stat(status);

/* Set up a CPU time timer with AST. */
        status = sys$setimr(C_EF,ten_ms,ast_rtn,rid2,TICK_TIMER);
        sys_stat(status);
/* Loop and count. */
       for(i=0;i<ITERATIONS;i++)
        {
                if((i%DISP_ITERATIONS) == 0)
                {
                        printf(" Loop: %d\n",i);
                }
        }
        printf("%d elapsed tics\n",rid1[CTR]);
        printf("%d cpu tics\n",rid2[CTR]);
```

Asynchronous System Traps Example (continued)

```
        return(SS$_NORMAL);
        }
        /* AST routine increments counter and resets timer. */
        int     ast_rtn(int     *timer_id)
        {
                int     status;

        /* Count appropriate 10ms tics. */
                timer_id[CTR]++;
        /* Reset CPU/elapsed timer. */
                status = sys$setimr(timer_id[EF],ten_ms,ast_rtn,
                                timer_id,timer_id[ID_ARG]);
                sys_stat(status);
                return(SS$_NORMAL);
        }

        $ cc ast_run
        $ link ast_run
        $ r ast_run
         Loop: 0
         Loop: 1000000
         Loop: 2000000
         Loop: 3000000
         Loop: 4000000
         Loop: 5000000
         Loop: 6000000
         Loop: 7000000
         Loop: 8000000
         Loop: 9000000
        233 elapsed tics
        230 cpu tics
$
```