OpenVMS KP Services

Doug Gordon

OpenVMS Engineering

What exactly are these things called anyway…?

- Originally Kernel Process Services.
- Misnamed from the beginning - not processes
- Now not exclusively kernel mode
- For a while, just “KPs”
- All procedure names still have KP

History

- Needed to emulate macro FORK in high level languages for C as a system language project.
- FORK only saves limited registers
- Difficult to tell C compiler about the limitations of FORK.
- KPS invented

What is it?

- Conceptually, it's a procedure that executes on its own stack[s].
- Procedure may be stalled and resumed.
- To the HLL compiler, stall and restart look like outbound procedure calls and the calls obey the calling standard.
- Restart causes the KP routine to resume at the return from the stall call.
- Once set up, all stack and saved register management is handled by the KP routines.

Why should you use KPs?

- Porting code that switches or manipulates stacks on Alpha or Vax to Itanium.
- Stack switch was a SMOP in Macro32 or Macro64
- IPF architecture is much more complicated.
  - Two stacks.
  - Asynchronous machine state.
  - Stack switch code is much more complicated and much harder to get right.
- IA64 assembler is not for the faint of heart

Why should you use KPs?

- One common solution, one set of bug fixes.
- IAS is not an optimizing assembler. Any optimization will be done in one place only.
- Supported interface
- Can be implemented and debugged on Alpha where tools are more mature
- VMS engineering is actively discouraging roll-your-own solutions.

Rules for Use

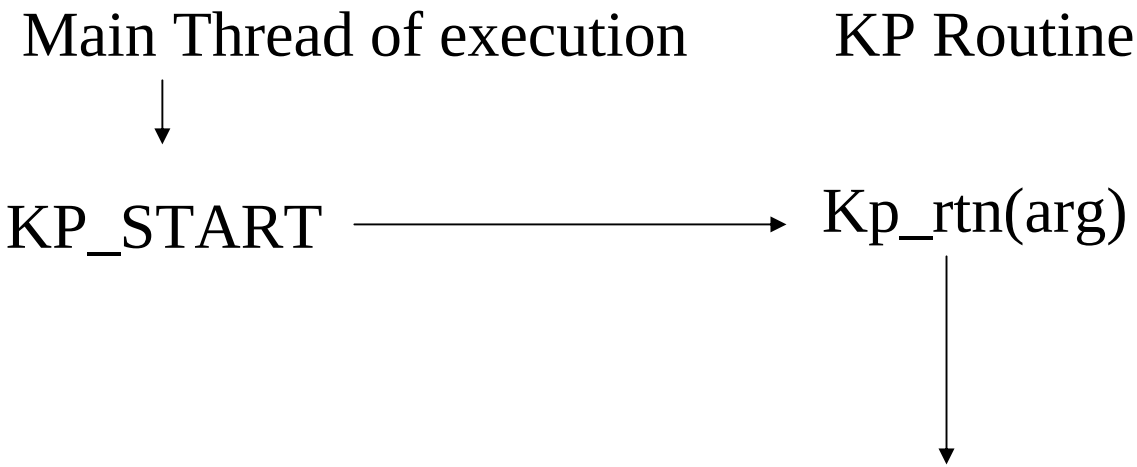
- KP routines only replace one [KESU] stack (or pair of stacks on IA64)
- All subsequent calls to KP routines must be made from the mode in which KP_START was called.
- KPB and stacks must be allocated from an appropriate region, at the appropriate mode, protection and ownership. (Details to follow)

Basic Routines

- EXE$KP_START - start a KP routine
- EXE$KP_STALL_GENERAL - stall a KP routine
- EXE$KP_RESTART - restart a stalled KP routine
- EXE$KP_END - terminate a KP routine and optionally clean up context.

EXE$KP_START

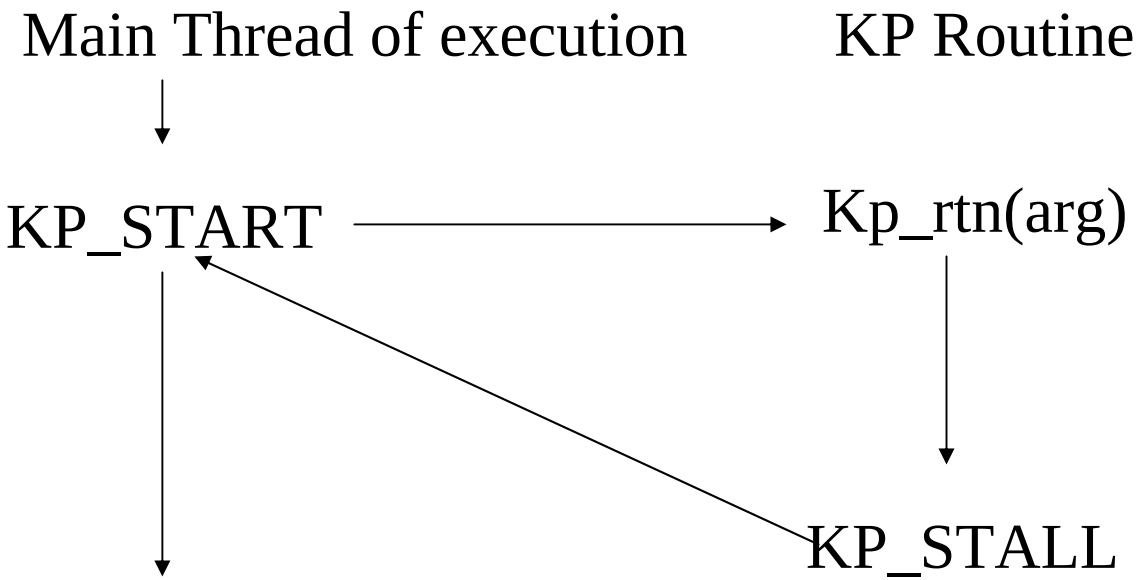
- Status = EXE$KP_START(kpb, routine, reg-mask)
- kpb - previously allocated and initialized
- routine - KP routine address
- reg-mask - register save mask (Alpha only. IA64 only supports the calling standard.)
- Suspend the current thread of execution, swap to the new stack[s] and call the specified routine. The only parameter to the called routine is the KPB.


Main Thread of execution
KP Routine
KP_START
Kp_rtn(arg)

EXE$KP_STALL_GENERAL

status = EXE$KP_STALL_GENERAL(kpb)

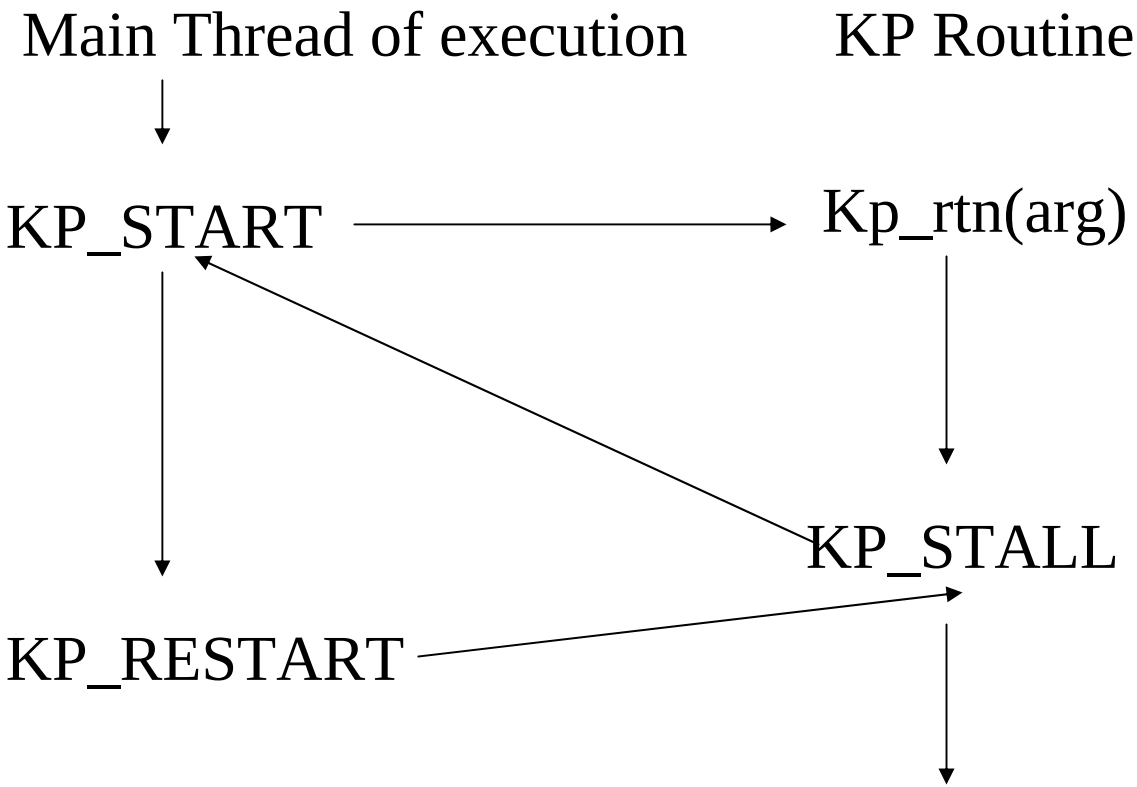
- Stall the current thread of execution, saving context onto the KP stack and returning to the most recent call that started or restarted this routine.
- The status returned from this routine is supplied by the routine that restarts this procedure.


Main Thread of execution
KP Routine
KP_START
Kp_rtn(arg)
KP_STALL

EXE$KP_RESTART

EXE$KP_RESTART(kpb [, thread_status])

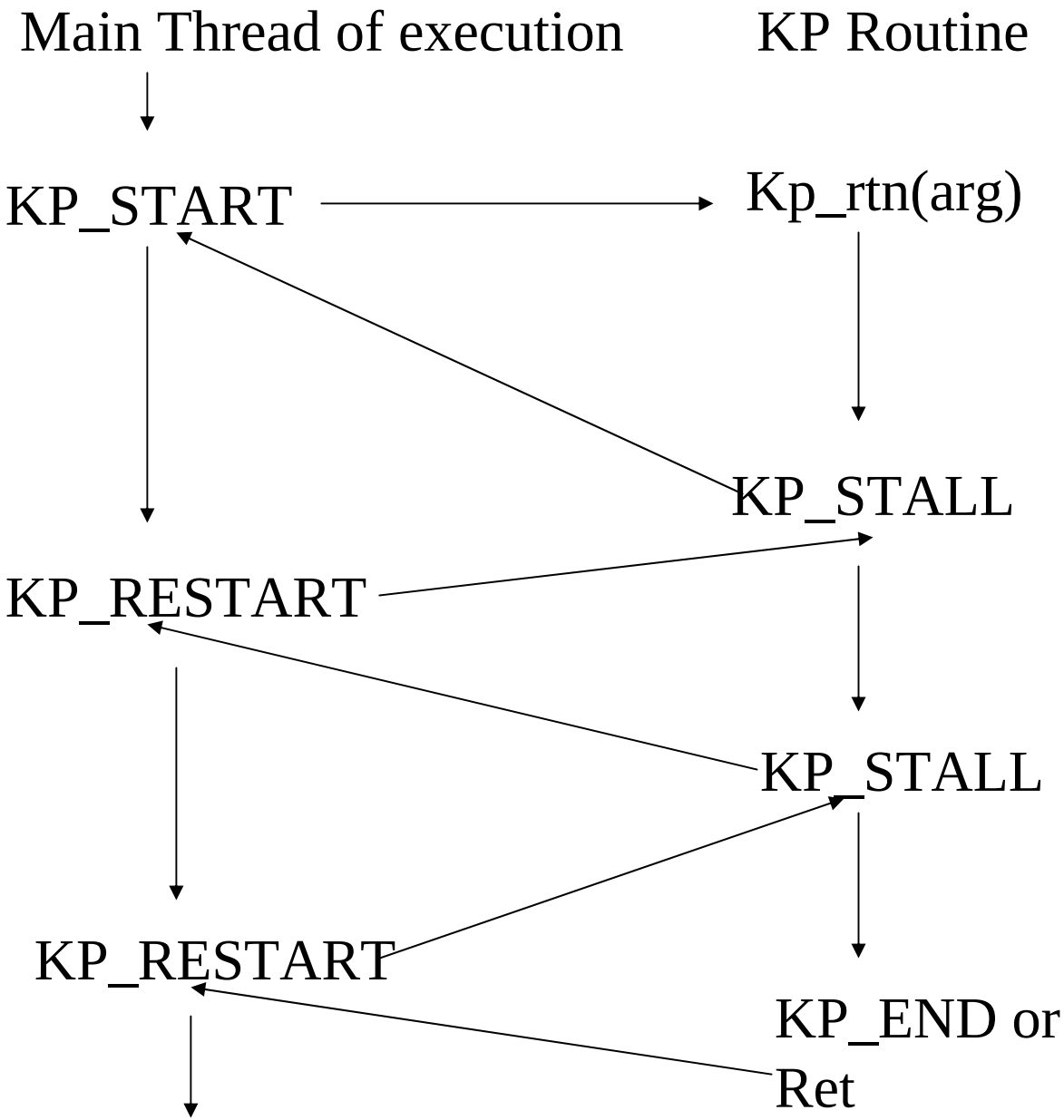
- Restart a previously stalled thread. Note that this may be a completely asynchronous operation with respect to the original thread of execution that started the KP routine.
- Thread_status, if provided, is used as the return status from EXE$KP_STALL_GENERAL. If omitted, SS$_NORMAL is returned.


Main Thread of execution
KP Routine
KP_START
Kp_rtn(arg)
KP_STALL
KP_RESTART

EXE$KP_END

status = EXE$KP_END(kpb [, status])

- Terminates the KP routine, returning control to the last thread of execution that started or restarted the KP routine.
- Status, if supplied, is passed to the optional routine specified by the end-rtn field of the KPB. SS$_NORMAL otherwise.
- RET from the KP routine calls KP_END automatically. In that case, return status from the KP procedure is used for the status argument.


Main Thread of execution
KP Routine
KP_START
Kp_rtn(arg)
KP_STALL
KP_RESTART
KP_STALL
KP_RESTART
KP_END or
Ret

KPBs

- A KPB is the data structure that holds the whole thing together.
- KPB represents all the context required to switch stacks and the procedure stacks themselves.
- Pointers to the base[s] of the stack[s] and the current SP on the non-active stack are stored in the KPB.
- KPB is semi-transparent. KP services maintain some fields. Many are directly or indirectly user-maintained.

KPBs cont.

- KPBDEF is in LIB, not STARLET
  - Code that switches stacks is not ordinary user code
- KPB consists of multiple sections.
  - Base section (required)
  - Scheduling
  - VMS Special Parameters
  - Spinlock
  - Debug
  - General Parameters

KPB - base section

Contains

- standard structure header
- stack sizes in bytes & stack base addresses
- flags (includes what areas are present.)
- saved memory stack pointer
- pointers to other optional areas.
- Other fields required by base routines

KPB - scheduling area

- Stall handling routine
- Restart handling routine
- Pointer to fork block
- End routine (required for user alloc call)
- A fork block

All except the end routine are primarily for driver-level code

KPB - other areas

- VEST - driver level IO code
- SPINLOCK - High IPL code
- DEBUG - limited tracing, limited support
- User Parameter - Application-defined

KPB - interesting flags

These may be set in the allocation calls:

- Area flags (VEST, SPLOC, DEBUG, PARAM)
- DEALLOC_AT_END
  - only if allocated via EXE$KP_ALLOCATE_KPB
- SAVE_FP - save **floating point** state (IA64 only)
- SET_STACK_LIMITS - call $SETSTK_64 at every stack switch
  - Correct stack limits are critical on IA64 for anything that signals, unwinds, or uses condition handlers.

Suggested Memory Allocation Attributes and Routines

Mode – Scope	KPB	Memory Stack	Register Stack
Kernel – System EXE$KP_ALLOC_KPB	Non-paged Pool KW EXE$ALONONPAGED	S0/S1 KW EXE$KP_ALLOC_MEM_STACK	S2 KW EXE$KP_ALLOC_RSE_STACK
Kernel – Process	Non-Paged Pool or P1 EXE$ALONONPAGED or EXE$ALOP1PROC	P1 – Permanent $CREATE_REGION/$CRETVA	P2 – Permanent KW EXE$KP_ALLOC_RSE_STACK_P2
Kernel – Image	P1 EXE$ALOP1IMAG	P1 – Non-permanent $CREATE_REGION/$CRETVA	P2 – Non-permanent KW
Exec – Process	P1 EXE$ALOP1PROC	P1 – Permanent $CREATE_REGION/$CRETVA	P2 – Permanent EW EXE$KP_ALLOC_RSE_STACK_P2
Exec – Image	P1 EXE$ALOP1IMAG	P1 – Non-permanent $CREATE_REGION/$CRETVA	P2 – Non-permanent EW
Super – Process	P1 EXE$ALOP1PROC	P1 – Permanent $CREATE_REGION/$CRETVA	P2 – Permanent SW EXE$KP_ALLOC_RSE_STACK_P2
Super – Image	P1 EXE$ALOP1IMAG	P1 – Non-permanent $CREATE_REGION/$CRETVA	P2 – Non-permanent SW
User – Image	P0 Heap/Malloc	P0 UW EXE$KP_ALLOC_MEM_STACK_USER	P2 – Non-permanent UW EXE$KP_ALLOC_RSE_STACK_P2

Note: Permanent regions survive image rundown. Permanent regions are not allowed in User mode.

Supplied Allocation Routines

- The system provides a set of convenience routines to aid in allocation of KPBs and stacks.
- EXE$KP_ALLOCATE_KPB(KPB_PPS kpb, int stack_size, int flags, int param_size)
- Kernel mode only. Same prototype as original Alpha routine.
- Note short pointer return for KPB to remain compatible with the previous implementation for device drivers written in C.
- On IA64, RSE stack size = memory stack size.
- Note, stack size is in BYTES. Will be minimized with KSTACKPAGES

Allocation Routines, continued

```
EXE$KP_USER_ALLOC_KPB(KPB_PPS kpb,
    int flags, int param_size, int (*kpb_alloc)(),
    int mem_stack_bytes, int (*memstk_alloc)(),
    int rse_stack_bytes, int (*rsestk_alloc)(),
    void (*end_rtn)())
```

Any-mode allocation with user-supplied allocation routines.

End routine required if clean up necessary.

System-supplied stack allocation routines are provided for most common needs.

Stack Allocation Routines

Most stack allocation routines have the same prototype:

status = routine(KPB_PQ, kpb, const int stack_pages)

- EXE$KP_ALLOC_MEM_STACK (kernel mode)
- EXE$KP_ALLOC_MEM_STACK_USER (user mode)
- EXE$KP_ALLOC_RSE_STACK (caller's mode)
- EXE$KP_ALLOC_RSE_STACK_P2 (caller's mode)
- EXE$KP_ALLOC_RSE_STACK_P2_ANY takes two additional parameters - region protection and access mode

- The first four calls may be used as the allocation routines required for EXE$KP_USER_ALLOC_KPB

Matching Deallocation Routines

- EXE$KP_DEALLOCATE_KPB
  - Only KPBs allocated by EXE$KP_ALLOCATE_KPB
  - Deallocates stack[s] and KPB
- EXE$KP_DEALLOC_MEM_STACK
- EXE$KP_DEALLOC_MEM_STACK_USER
- EXE$KP_DEALLOC_RSE_STACK
- EXE$KP_DEALLOC_RSE_STACK_P2
  - Handles RSE stacks allocated by EXE$KP_ALLOC_RSE_STACK_P2 or EXE$KP_ALLOC_RSE_STACK_P2_ANY

Useful files

- KPBDEF.*
  - Structure definitions
- KPSTACKDEF.* (IA64 only)
  - stack layout for the saved context
- EXE_ROUTINES.H
  - C prototypes
- IODEF.STB
  - Symbols for SDA

Consumers in the base OS

- F11XQP
- RMS (exec mode)
- DCL (supervisor mode)
- DDTM
- DECnet Phase V
- DECdns
- TCP/IP
- MTAACP

RMS and the XQP run this way on Alpha as well. They were the guinea pigs for proof of concept.

Example, DCL

- On VAX and Alpha, DCL saves supervisor stack state at initialization of the process and at the beginning of the command loop simply resets the stack to the saved state. This is much harder on IA64.
- Now the DCL main command loop runs as the KP process. When an image is run, DCL stalls the command loop and returns to the original stack. In that context, an REI call is performed to switch to outer mode which resets the supervisor mode stack to base.
- DCL's super mode exit handler takes control at image exit and the command loop is resumed via KP_RESTART.

For more info on the original KP implementation see:
OpenVMS AXP Internals and Data Structures - Chapter 5
Writing OpenVMS Alpha Device Drivers in C

Note: Although the documentation on the original KP implementation states that KPs can only be used in kernel mode at IPL 3 or higher, the current implementation does not impose these restrictions. KP services may now be used in any mode at any IPL.