Memory Protection Keys

Memory Protection Keys for Userspace (PKU aka PKEYs) is a feature which is found on Intel's Skylake (and later) "Scalable Processor" Server CPUs. It will be available in future non-server Intel parts and future AMD processors.

For anyone wishing to test or use this feature, it is available in Amazon's EC2 C5 instances and is known to work there using an Ubuntu 17.04 image.

Memory Protection Keys provides a mechanism for enforcing page-based protections, but without requiring modification of the page tables when an application changes protection domains. It works by dedicating 4 previously ignored bits in each page table entry to a "protection key", giving 16 possible keys.

There is also a new user-accessible register (PKRU) with two separate bits (Access Disable and Write Disable) for each key. Being a CPU register, PKRU is inherently thread-local, potentially giving each thread a different set of protections from every other thread.

There are two new instructions (RDPKRU/WRPKRU) for reading and writing to the new register. The feature is only available in 64-bit mode, even though there is theoretically space in the PAE PTEs. These permissions are enforced on data access only and have no effect on instruction fetches.

Syscalls

There are 3 system calls which directly interact with pkeys:

```
int pkey_alloc(unsigned long flags, unsigned long init_access_rights)
int pkey_free(int pkey);
int pkey_mprotect(unsigned long start, size_t len,
                  unsigned long prot, int pkey);
```

Before a pkey can be used, it must first be allocated with pkey_alloc(). An application calls the WRPKRU instruction directly in order to change access permissions to memory covered with a key. In this example WRPKRU is wrapped by a C function called pkey_set().

```
int real_prot = PROT_READ|PROT_WRITE;
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE);
ptr = mmap(NULL, PAGE_SIZE, PROT_NONE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);
ret = pkey_mprotect(ptr, PAGE_SIZE, real_prot, pkey);
... application runs here
```

Now, if the application needs to update the data at 'ptr', it can gain access, do the update, then remove its write access:

```
pkey_set(pkey, 0); // clear PKEY_DISABLE_WRITE
*ptr = foo; // assign something
pkey_set(pkey, PKEY_DISABLE_WRITE); // set PKEY_DISABLE_WRITE again
```

Now when it frees the memory, it will also free the pkey since it is no longer in use:

```
munmap(ptr, PAGE_SIZE);
pkey_free(pkey);
```

> **Note**
>
> pkey_set() is a wrapper for the RDPKRU and WRPKRU instructions. An example implementation can be found in tools/testing/selftests/x86/protection_keys.c.

Behavior

The kernel attempts to make protection keys consistent with the behavior of a plain mprotect(). For instance if you do this:

```
mprotect(ptr, size, PROT_NONE);
something(ptr);
```

you can expect the same effects with protection keys when doing this:

```
pkey = pkey_alloc(0, PKEY_DISABLE_WRITE | PKEY_DISABLE_READ);
pkey_mprotect(ptr, size, PROT_READ|PROT_WRITE, pkey);
something(ptr);
```

That should be true whether something() is a direct access to 'ptr' like:

```
*ptr = foo;
```

or when the kernel does the access on the application's behalf like with a read():

```
read(fd, ptr, 1);
```

The kernel will send a SIGSEGV in both cases, but si_code will be set to SEGV_PKERR when violating protection keys versus SEGV_ACCERR when the plain mprotect() permissions are violated.