第二十一章　实验 3: 用户进程与异常处理

21.1　简介

用户进程是操作系统对在用户模式运行中的程序的一种抽象。之前的实验在 ChCore 中实现了操作系统启动和正常执行的一些必要功能，本实验将对 ChCore 的功能进行加强，使其能够支持用户进程，并处理用户进程执行过程中产生的异常和系统调用。

具体而言，本实验的包括三个部分：

- 第一部分：使用对应的数据结构，支持创建、启动并管理用户进程与线程
- 第二部分：熟悉 AArch64 的异常处理流程，为 ChCore 添加异常处理的支持，正确处理用户程序所触发的异常
- 第三部分：正确处理一系列关键系统调用与异常，保证用户程序的正常执行与退出

21.1.1　评分

在开始本实验前，请先使用如下命令编译整个操作系统和所有用户程序：

```
chcore$ make
```

本实验中，代码部分的总成绩为 100 分。可使用如下命令检查当前得分：

```
chcore$ make grade
...
Score: 100/100
```

如需单独运行某一个用户程序X以进行调试，可以使用如下命令：

```
chcore$ make run-X
```

其中，X 代表user/lab3目录下特定的用户程序的名称，如make run-hello即为运行名为hello的程序。此外，还可以使用make run-X-gdb与make gdb组合的方式，使用 GDB 对运行用户程序 X 的 ChCore 内核进行调试。

21.2 第一部分：实现用户进程

21.2.1 AArch64 中的异常级别

AArch64 体系结构使用“异常级别”这一概念定义程序执行时所拥有的特权级别。在 AArch64 中定义了四个异常级别，从低到高依次为 EL0、EL1、EL2 和 EL3。关于每个异常级别的具体含义和常见用法，请参见第 二章。ChCore 仅使用了其中的两个异常级别：EL0 和 EL1。其中，EL1 代表 ChCore 中的内核模式，kernel/目录下的所有操作系统内核代码均运行于此异常级别。EL0 代表用户模式，user/目录下的用户库函数代码与用户程序代码均以用户进程的抽象运行于此异常级别下。

21.2.2 基于 Capability 的资源访问控制

在 ChCore 中，公开给应用程序的所有内核资源均采用**能力（Capability）**机制进行管理。所有的内核资源（如物理内存等）均被抽象为了**内核对象（kernel object）**，应用程序通过整型的标识符 Capability 访问从属于该进程的内核对象。为方便理解，可以类比 Linux 中的文件描述符。在 Linux 中，文件描述符（fd）即相当于一个整型的 Capability，而文件本身则相当于一个内核对象。对于同一进程中的所有线程，其所能访问的内核对象以及 Capability 与内核对象的映射关系完全相同。关于 Capability 机制的更多详细介绍，请参照：链接

```c
// 分配 Cap 和内核对象的相关代码（错误处理的相关代码已被省略）
// 此函数中分配了一个 PMO 内核对象，并且返回了对应的 Cap 以
// 便后续访问这一内核对象
int sys_create_pmo(u64 size, u64 type)
{
  int pmo_cap;
  struct pmobject *pmo;

  // 分配一个内核对象，类型为 TYPE_PMO，大小为 sizeof(*pmo)
  pmo = obj_alloc(TYPE_PMO, sizeof(*pmo));
  // 完成内核对象初始化的相关工作
  pmo_init(pmo, type, size, 0);
  // 为访问这一内核对象分配对应的 Cap
  pmo_cap = cap_alloc(current_process, pmo, 0);
  return pmo_cap;
}

// 使用 Cap 查找并访问对应的内核对象的相关代码（错误处理
// 的相关代码已被省略）
// 使用上述系统调用 sys_create_pmo 返回的 Cap 值，查询并访
// 问对应的 PMO 对象
int sys_map_pmo(u64 process_cap, u64 pmo_cap, u64
  ↪  addr, u64 perm)
{
  struct pmobject *pmo;

  // 使用 TYPE（TYPE_PMO）和 Cap 查询对应的 PMO 对象
  pmo = obj_get(current_process, pmo_cap, TYPE_PMO);
  // Use pmo
  ...

  // 访问 PMO 对象结束，使用 obj_put 重设原有 Cap 对应的
  // PMO 对象。
  // 这一代码是为了保证并发状态下的正确性而准备的，无需关注
  // 相关细节，只需要了解如何修改一个 Cap 所对应的对象即可。
  // 注：这里，pmo 这一对象所对应的 cap 值存储于 PMO 这一内核
  // 对象的结构体内部
  obj_put(pmo);
  ...
}
```

代码片段 21.1: Capability 示例代码

虽然本实验无需实现任何 Capability 机制的相关逻辑，但仍需基本了解分配 Capability、关联 Capability 与内核对象、使用 Capability 查询管理内核对象等相关接口。代码片段 21.1中以名为sys_create_pmo和sys_map_pmo的两个函数为例，对相关机制进行了讲解。其中，**物理内存对象（Physical Memory**

Object，PMO）是代表一块物理内存的内核对象。

详细的 Capability 相关代码位于kernel/process/capability.h中，包括了分配、拷贝、删除、回收 Capability 的内容，有兴趣的读者可以自行探索相应内容的设计和实现。

21.2.3 线程和进程

ChCore 内核目前使用process结构体表示**进程**，使用thread结构体表示**线程**。ChCore 的一个进程是一些对特定内核对象享有相同的 Capability 的线程的集合。与 Linux 一样，ChCore 中的每个进程可能包含多个线程，而每个线程则从属且仅从属于一个进程。

头文件kernel/process/process.h定义了struct process，如代码片段 21.2所示。

```
1 struct process {
2   struct slot_table slot_table;
3   struct list_head thread_list;
4 };
```

代码片段 21.2: 进程数据结构

其中，各成员的含义如下：

- slot_table：该进程有权访问的内核对象的数组（内核对象在数组中的索引即为该对象的 cap）。
- thread_list：从属于该进程的所有线程的链表。

头文件kernel/process/thread.h定义了struct thread，如代码片段 21.3所示。

```
1 struct thread {
2   struct list_head    node;
3   struct thread_ctx   *thread_ctx;
4   struct vmspace      *vmspace;
5
6   struct process      *process;
7 };
```

代码片段 21.3: 线程数据结构

其中，各成员的含义如下：

- node：从属于同一进程的线程的链表。
- thread_ctx：该线程的上下文。目前，其中仅存储属于当前线程的所有寄存器的值和当前线程的类型，后续实验 4 会向其中添加更多同调度相关的内容。
- vmspace：该线程所创建的虚拟内存映射的相关信息
- process：该线程所从属于的进程

在实验 3 中，我们仅会创建一个包含单线程的用户进程，并仅启用一个 ID 为 0 的 CPU 来运行该进程。与调度程序和多 CPU 相关的内容将在实验 4 中实现。

21.2.4 创建并执行主线程

截止目前，ChCore 中进程与线程相关的基本概念已介绍完毕。本节将对用户进程的创建流程和内存结构进行介绍。

由于目前 ChCore 中尚无文件系统，所有二进制用户程序镜像将以静态链接的方式，被直接嵌入到内核镜像中，以便用户进程进行加载和运行。ChCore 在 **cpio** 的帮助下将所有user/目录中编译出的 ELF 文件合并到了一个文件中，并将该文件通过变量binary_cpio_bin_start嵌入内核。更多细节可查看user/binary_include.S和scripts/compile_user.sh中的内容。

用户进程创建流程

在 ChCore 中，创建一个初始进程并将正确的 ELF 文件载入到进程中开始执行的代码位于kernel/main.c的main()函数中。以下为从操作系统初始化到调用第一行用户代码为止的代码调用图:

1. start
2. main (kernel/main.c)
   (a) uart_init
   (b) mm_init
   (c) exception_init

(d) process_create_root

i. process_create

ii. thread_create_main

(e) eret_to_thread

i. switch_context

在上述调用图中，函数eret_to_thread最为核心。该函数使用 **eret 指令**，完成了从内核模式到用户模式的切换，并在用户模式下开始运行用户代码。如《Arm 架构参考手册》 [2] 所述，eret指令的语义主要包括：

- 检查当前异常级别的 SPSR 寄存器（在本例中为 SPSR_EL1）中的值，以选择在eret指令执行完成后处理器应处于的目标异常级别（在本例中为 EL0）。
- 将当前异常级别 SPSR 寄存器中的值设置到 PSTATE 相关的一系列寄存器中。
- 将当前异常级别 ELR（异常返回地址指针寄存器，详见后文，本例中为 ELR_EL1）中的内容设置到 PC 寄存器中。
- 切换到目标异常级别并开始执行。切换完成后，处理器将会切换为采用目标异常等级的 SP 寄存器（在本例中为 SP_EL0）作为栈寄存器进行执行。

用户进程内存结构

此外，一个用户线程在运行时，主要使用了以下内存区域：

- 用户栈：在thread_create_main中创建，是用户进程在用户模式运行时用作栈的一小段内存。
- 内核栈：在create_thread_ctx中创建，是用户进程在内核模式下运行时用作当前线程栈的一小段内存。struct thread_ctx即存储在当前线程的内核栈上。
- 代码/数据区域：在load_binary中创建，是用于存储从 ELF 文件中加载出来的代码和数据段的一系列内存区域。

- 用户堆：由sys_handle_brk系统调用创建，是用作用户进程的堆的一小段内存。该部分内存的实际物理内存分配将在第一次访问时通过缺页异常处理完成。

后续练习将基于上述所有基础知识，完成用户进程的创建与执行。

练习 1

在kernel/process/thread.c、kernel/sched/context.c和kernel/sched/sche
请补全以下函数，以实现对第一个用户进程和线程的创建和执行：

- load_binary：解析 ELF 文件，并将其内容加载到新线程的用户内存空间中。
- init_thread_ctx：初始化线程的上下文，以便启动当前线程。
- switch_context：切换到当前线程的上下文。

练习 2

请简要描述process_create_root这一函数所的逻辑。**注意**：描述中需包含thread_create_main函数的详细说明，建议绘制函数调用图以描述相关逻辑。

完成上述内容后即可使用make run-hello命令在 ChCore 中执行hello.bin这一用户程序。如无意外，ChCore 应能够正确地为该程序创建对应的进程，进入用户模式并开始执行该程序。

然而，目前hello.bin并不能输出hello world并正常退出。这是由于该程序使用了svc #0指令进行printf相关的系统调用。由于 ChCore 尚未配置从用户模式（EL0）切换到内核模式（EL1）的相关内容，在尝试执行svc指令时，ChCore 将根据目前的配置（尚未初始化，异常处理向量指向随机位置）执行位于随机位置的异常处理代码，进而导致触发错误指令异常。同样的，由于错误指令异常仍未指定处理代码的位置，对该异常进行处理会再次出发错误指令异常。ChCore 将不断重复此循环，并最终显示为 QEMU 不响应。后续的练习中将会通过正确配置异常向量表的方式，对这一问题进行修复。

为了在用户程序无法输出的前提下检查第一部分的实现是否正确，请使用命令make run-hello-gdb和命令make gdb的组合，将 GDB 连接到运行着 hello 程序的 qemu 模拟器上。进入 GDB 后，可使用b START在用户程序的入口点创建断点，并使用continue命令继续 QEMU 的执行。如果在用户

程序入口点处的断点被触发，则意味着第一部分内容已正确完成，ChCore 成功进入了用户模式，并开始执行用户程序的第一条指令了。

21.3 第二部分：异常处理

截止目前，由于 ChCore 尚未对用户模式与内核模式的切换进行配置，一旦 ChCore 进入用户模式执行就再也无法返回内核模式使用操作系统提供其他功能了。在这一部分中，我们将通过正确配置异常向量表的方式，为 ChCore 添加异常处理的能力。

在开始这一部分前，请先阅读《ARMv8-A 程序员指南》 [1]（链接）中的第 10 章“ARM 异常处理”的相关内容，以便基本了解 AArch64 中的异常处理流程。后文也将选取该指南和《Arm 架构参考手册》[2]（链接）中的相关内容，采用 AArch64 中对**异常（Exception）**、**中断（Interrupt）**、**中止（Abort）**等的术语的定义，对 AArch64 中异常处理的流程进行简要介绍。

注意：为了精准描述 AArch64 硬件的特性，后文中一些表述翻译自《ARMv8-A 程序员指南》 [1] 与《Arm 架构参考手册》 [2]。为保证阅读连贯性，将不会在每次涉及对应内容时进行标注，请谅解。

21.3.1 AArch64 中的异常

在 ARM 术语中，异常是指低特权级软件（如用户程序）请求高特权软件（例如内核中的异常处理程序）采取某些措施以确保程序平稳运行的系统事件，包含**同步异常**和**异步异常**：

- 同步异常：通过直接执行指令产生的异常，其异常返回地址处的指令同异常发生的原因存在关联。
- 异步异常：与正在执行的指令无关的异常。

同步异常的来源包括**同步中止（synchronous abort）**和**异常生成指令**。当直接执行一条指令时，若取指令或数据访问过程失败，则会产生同步中止。异常生成指令则是指可以主动生成异常的指令，此类指令（包括 svc，hvc 和 smc）通常被用户用于主动制造异常以请求高特权级别软件提供服务（如系统调用）。

异步异常的来源包括普通中断 **IRQ**、快速中断 **FIQ** 和系统错误 **SError**。IRQ 和 FIQ 是由其他与处理器连接的硬件产生的中断，系统错误则包含多种

可能的原因。本实验并不涉及异步异常的处理，关于异步异常的更多详细信息，请感兴趣的读者自行参考手册中的相关内容。

21.3.2 异常向量表

发生异常后，处理器需要找到与发生的异常相对应的异常处理程序代码并执行。在 AArch64 中，存储于内存之中的异常处理程序代码被叫做**异常向量（exception vector）**，而所有的异常向量被存储在一张**异常向量表（exception vector table）**中。

AArch64 中的每个异常级别都有其自己独立的异常向量表，其虚拟地址由该异常级别下的异常向量基地址寄存器（VBAR_EL3，VBAR_EL2 和 VBAR_EL1）决定。每个异常向量表中包含 16 个条目，每个条目里存储着发生对应异常时所需执行的异常处理程序代码。

发生异常时，使用异常向量表中的哪个条目，取决于以下因素：

- 异常类型，即该异常是 SError，FIQ，IRQ、同步异常中的哪一个。
- 如果异常是在相同的异常级别上触发的，则异常处理程序将要使用的栈指针会对选择的表项产生影响（例如，在 EL2 中触发异常时，异常处理程序会根据 SPSel 寄存器的值决定使用 EL0（SP_EL0）还是 EL2（SP_EL2）的栈指针，而这一选择同样决定了需要使用的异常向量条目）。
- 如果异常是在较低的异常级别上触发的，则低异常级别是处理器的执行状态（AArch64 或 AArch32）会对选择的表项产生影响。

表 21.1中列出了每个异常向量条目的偏移量。

21.3.3 异常处理相关寄存器

本实验仅专注于页错误、系统调用等同步异常的处理。异步异常的配置与处理等相关内容将会在后续实验 4 中有所涉及。

处理同步异常时，处理器使用一系列寄存器进行该同步异常相关的必要信息的保存，包括：

- **异常症状寄存器（Exception Syndrome Register）**ESR_ELx：存储有关异常原因的信息。有关此寄存器中各个位的含义，请参阅《ARMv8 程序员指南》。

表 21.1: VBAR_ELx 指向的异常向量表 [2]

地址	异常类型	异常发生时处理器状态
+ 0x000	同步异常	ELx 使用 SP_EL0 作为 SP
+ 0x080	IRQ	
+ 0x100	FIQ	
+ 0x180	SError	
+ 0x200	同步异常	ELx 使用 SP_ELx 作为 SP
+ 0x280	IRQ	
+ 0x300	FIQ	
+ 0x380	SError	
+ 0x400	同步异常	EL0 运行于 AArch64 状态
+ 0x480	IRQ	
+ 0x500	FIQ	
+ 0x580	SError	
+ 0x600	同步异常	EL0 运行于 AArch32 状态
+ 0x680	IRQ	
+ 0x700	FIQ	
+ 0x780	SError	

- **错误地址寄存器（Fault Address Register）**FAR_ELx：存储所有同步指令中止、同步数据中止和对齐异常所对应的虚拟地址。如发生缺页异常时，触发该异常的页的虚拟地址即存储在该寄存器中。

- **异常链接寄存器（Exception Link Register）**ELR_ELx：该异常的首选返回地址。对于某些同步异常（例如 SVC），它指向异常生成指令的下一条指令的地址。对于其他的同步异常，它指向发生异常的指令，以便于重新执行。对于由中断等导致的异步异常，ELR_ELx 指向尚未执行或未完全执行的第一条指令的地址。

注意：异步异常发生原因等信息将保存在**通用中断控制器（Generic Interrupter Controller，GIC）**内的寄存器中，而非 ESR_ELx 中。

21.3.4 综合示例

本小节将会给出综合上文的示例，以便理解整个处理过程。假设 AArch64 处理器正在用户线程中执行代码，并且遇到了一条指令集中未定义的指令。此时，处理器会发生未定义指令异常，并执行如下主要操作：

1. 处理器将异常原因放入 ESR_EL1 中，并将返回地址（即未定义指令的地址）放入 ELR_EL1 中。

2. 处理器检查 VBAR_EL1 以获得 EL1 中使用的异常向量表的地址。由于当前异常是来自 AArch64 模式中 EL0 特权级的同步异常，因此处理器将选择条目 VBAR_EL1+0x400。

3. 处理器将特权级切换到 EL1。这一过程中，包含了如保存 PSTATE、使用 SP_EL1 作为栈指针等内容，从而完成了从用户栈到内核栈的切换。

4. 处理器执行 VBAR_EL1+0x400 处的代码，在 ChCore 中，这是一条跳转到异常处理程序的b指令。

21.3.5 为 ChCore 设置异常向量表

在 ChCore 中，仅使用了 EL0 和 EL1 两个异常级别，因此仅需要对 EL1 异常向量表进行初始化即可。在本实验中，ChCore 内除系统调用外所有的同步异常均交由`handle_entry_c`函数进行处理。遇到异常时，硬件将根据

ChCore 的配置执行对应的汇编代码，将异常类型和当前异常处理程序条目类型作为参数传递，并最终调用`handler_entry_c`使用 C 代码处理异常。

练习 3

完善异常处理，需要阅读与修改`kernel/exception/`下的`exception_table.s`、`exception.S`和`exception.c`。需要修改的内容包括：

- 修改`exception_table.S`的内容，可借助该文件中的某些宏，填写异常向量表。
- 完成`exception.c`中的`exception_init`函数，使得 kernel 启动后能够正确设置异常向量表。
- 修改异常处理函数，使得当发生**异常指令异常**时，让内核使用 kinfo 打印在`esr.h`中定义的宏`UNKNOWN`的信息，并调用`sys_exit`函数中止用户进程。

练习 3 完成后，即可使用`user/lab3`目录下的一些测试程序来测试异常处理。其中，`badinsn`与`badinsn2`两个测试程序会在进行任何系统调用之前触发异常，因此，可以使用`make run-badinsn`和`make run-badinsn2`来检查异常处理的相关配置是否正确。此外，可以使用`make grade`指令进行评分，查看是否能够成功通过这两个测试用例。

21.4 第三部分：系统调用和缺页异常

21.4.1 系统调用

系统调用是用户程序请求操作系统帮助其提供一些需要高特权级方可完成的任务的方式。通常而言，为了实现系统调用，用户程序需要在用户模式下使用一些指令，主动产生异常，从而进入内核模式。在内核模式下，首先，操作系统代码和硬件将共同保存用户程序的状态。之后操作系统将选取并执行该系统调用所对应的代码，完成系统调用的实际功能，并保存返回结果。最后，操作系统和硬件将再次协作，共同恢复用户程序的状态，将系统调用的返回结果告知用户程序，并继续用户程序的执行。

练习 4

和其他异常不同，ChCore 中的系统调用是通过使用汇编代码直接跳转到`syscall_table`中的相应条目来处理的。请阅读`kernel/exception/exception_table.S`中的代码，并简要描述 ChCore 是如何将系统调用从异常向量分派到系统调用表中对应条目的。

练习 5

在`user/lib/syscall.c`中完成`syscall`这一用户库函数，在其中使用`SVC`指令进入内核态并执行相应的系统调用。在此过程中，需要使用到 GCC 内联汇编的相关内容，请参考“GCC 官方文档”（链接）[a]。

[a]https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html

练习 6

在 ChCore 中完成以下系统调用。

- `sys_putc`：`printf`所使用的基本系统调用，需要使用`uart_send`标准输出一个字符
- `sys_exit`：具有退出当前用户线程的功能，需要将对应编号的系统调用分派到`sys_exit`这一函数上。
- `sys_create_pmo`：用于测试缺页异常，实现在`vm_syscall.c`。
- `sys_map_pmo`：用于测试缺页异常，实现在`vm_syscall.c`。
- `sys_handle_brk`：用户线程将使用`sys_handle_brk`创建或扩展用户堆。通过这一系统调用，当前进程的堆将被扩大至虚拟地址 addr。更多详细信息请参见`/kernel/mm/vm_syscall.c`的注释。

然而，截止目前，ChCore 仍无法正常完成任何用户程序的执行。

用户程序所对应的 ELF 文件的入口函数是在`user/CMakeList.txt`指定，并由第一部分实现的`load_binary`和`init_thread_ctx`函数加载到线程初始的程序计数器中去的。用户程序的入口点为`user/lib/libmain.S`中的`START`函数，该函数负责调用`_start_c`函数，并在其中最终调用用户程序

定义的 main 函数。

然而，目前当 START 函数结束后，将导致**指令中止（instruction abort）**。

练习 7

请使用 GDB 检查START函数执行结束后程序计数器的值，解释为何会发生这一现象，并尝试在 ChCore 的异常处理器中处理对应类型的异常。

练习 8

在合适的地方添加对`sys_exit`的调用，使用户主线程能够正常退出。一般而言，退出当前线程后，内核中的调度器会将当前线程移出调度队列，并继续从调度队列中选择下一个可执行的线程进行执行。然而，由于目前 ChCore 中仅包含一个线程，因此，在唯一的用户线程退出后，ChCore 将中止内核的运行并直接退出。后续实验会对这一问题进行修复。

完成上述内容后，请使用`make run-hello`命令在 ChCore 中执行`user/lab3/hello.c`程序。现在，该程序应当能够正确在控制台上打印出“Hello world”这一信息并正常退出（表现为 ChCore 在输出`sys_exit`后直接退出）。此外，也应当能够通过`testputc`、`testcreatepmo`、`testmappmo`和`testmappmoerr`的测试。

21.4.2 处理缺页异常

页错误（错误码 36 或 37）在操作系统中起到了重要的作用，为**延迟内存分配（lazy memory allocation）**和内存保护等功能提供了重要支持。当 AArch64 的处理器发生缺页异常时，它会将发生错误的虚拟地址存储于 FAR_ELx 寄存器中，并异常处理流程。在`kernel/exception/pgfault.c`中，本实验提供了缺页异常处理函数`do_page_fault()`的一部分。

练习 9

完成缺页异常处理，具体而言，在函数`handle_entry_c`中将缺页异常转发给函数`do_page_fault()`处理，并完成`do_page_fault()`和`handle_trans_fault()`函数。

至此，ChCore 应能够成功通过`faultread`、`faultwrite`、`testpf`和`testsbrk`四个测试用例，并在`make grade`中获得全部分数。

参考文献

[1] ARM. Arm cortex-a series programmer's guide for armv8-a. https://developer.arm.com/documentation/den0024/a, 2015.

[2] ARM. Arm architecture reference manual. https://static.docs.arm.com/ddi0487/fa/DDI0487F_a_armv8_arm.pdf?_ga=2.181644388.2107974726.1583153879-1487747685.1581514464, 2020.

实验 ***3***：扫码反馈