|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 武汉大学国家网络安全学院教学实验报告 | | | | | |
| 课程名称 | 操作系统设计与实践 | | 实验日期 | | 2024.11.10 |
| 实验名称 | 基本实验环境搭建 | | 实验周次 | | 第一周 |
| 姓名 | 学号 | | 专业 | | 班级 |
| 王亚鹏 | 2022302181161 | | 信息安全 | | 5班 |
| 杨依磊 | 2022302181159 | | 信息安全 | | 5班 |
| 杜泓波 | 2022302181162 | | 信息安全 | | 5班 |
| 侯名扬 | 2022302181165 | | 信息安全 | | 5班 |
| 1. 实验目的及实验内容   （要求掌握的知识；实验内容；原理分析） | | | | | |
| **总目标：**  **多进程的实现机理与进程调度**  **实验内容：**  1. 多进程问题，如何扩展单进程到多进程，如何扩展中断支持多进程？  2. 如何实现系统调用  3. 进程调度问题，弄清楚实现调度的基本思路  **原理分析：**  **1. 多进程原理**  多进程系统允许操作系统在同一时刻运行多个进程。每个进程在内存中拥有独立的地址空间和资源，系统通过时间片轮转的方式在各进程之间切换。多进程的核心原理在于操作系统内核通过进程控制块（PCB）来管理每个进程的状态，并提供中断和上下文切换的机制，以便在多个进程间进行调度和管理。   * **进程控制块 (PCB)**：PCB存储了进程的基本信息，包括进程ID、状态、程序计数器、寄存器值、优先级等。在多进程环境中，操作系统通过PCB来区分不同的进程，并根据调度算法选择合适的进程执行。 * **中断处理**：操作系统通过硬件中断（如时钟中断）实现多进程管理。在中断发生时，操作系统保存当前进程的上下文信息（如寄存器值），并通过调度算法选择新的进程来执行。这一过程称为上下文切换。   **2. 系统调用原理**  系统调用是用户进程与操作系统交互的接口，用户进程通过系统调用请求操作系统提供特权服务（如文件操作、内存分配、进程管理等）。系统调用通过软中断的方式将用户进程的控制权转交给操作系统内核，并在系统调用执行完毕后将控制权交还给用户进程。   * **系统调用表**：系统调用表中保存了所有系统调用的入口地址。操作系统根据系统调用号查找对应的函数地址，执行用户进程请求的操作。 * **系统调用流程**：在用户态调用系统调用时，系统通过中断向量表找到对应的中断处理程序，保存用户态信息，切换到内核态执行请求的操作，然后再恢复用户态的上下文。   **3. 进程调度原理**  进程调度是多任务系统的关键技术之一，操作系统需要根据一定的算法在多个进程间分配CPU资源，以实现系统资源的高效利用。进程调度的原理主要包括以下几点：   * **调度算法**：常见的调度算法包括优先级调度、时间片轮转、多级反馈队列等。本次实验中，多级反馈队列调度算法被用于模拟进程的调度。多级反馈队列是一种自适应的调度算法，它根据进程的响应时间和使用时间，将进程动态调整到不同的队列中，以便对不同类型的任务（如CPU密集型和I/O密集型）进行优化调度。 * **时钟中断**：在操作系统中，时钟中断用于周期性地触发调度器。时钟中断发生时，系统保存当前进程的状态，并调用调度程序决定是否切换到另一个进程。这一机制使得系统能够实现时间片轮转，保障所有进程都能公平地获得CPU时间。   **4. 多级反馈队列调度算法**  多级反馈队列调度算法是一种将进程根据优先级分层的调度方式。它使用多个队列，每个队列对应一个优先级，并且随着进程的运行和等待情况动态调整其所在的队列。该算法有以下特性：   * **自适应调度**：根据进程的CPU使用情况，系统会自动将进程调整到不同的队列中。短时间内占用CPU的进程会保留较高优先级，而长时间占用CPU的进程则会逐渐降低优先级。 * **抢占机制**：高优先级队列中的进程总是优先调度。如果一个新进程进入更高优先级的队列，系统会中断当前的低优先级进程，转而执行新进程。 * **时间片动态调整**：不同优先级的队列可以设置不同的时间片，从而在高优先级队列中实现更短的响应时间，而在低优先级队列中实现更长的连续执行时间。   **5. 进程管理的基本过程**  进程管理包括进程的创建、销毁、切换、阻塞和唤醒等操作。在多进程环境中，进程的创建和销毁会对系统资源（如内存、CPU、I/O等）进行动态管理。系统通过初始化多进程控制块、扩展局部描述符表（LDT）和任务状态段（TSS）等来支持多进程调度和管理。  **6. 时钟中断的改进**  为了支持多进程管理，本实验要求对时钟中断进行修改。通过修改时钟中断的处理函数，使得系统在时钟中断发生时自动进行调度判断，从而支持多任务的时间片轮转 | | | | | |
| 1. 实验环境及实验步骤   （本次实验所使用的器件、仪器设备等的情况；具体实验步骤） | | | | | |
| 实验环境：  **X86 32位Linux环境**  实验步骤：   1. **多进程问题，如何扩展单进程到多进程，如何扩展中断支持多进程？**   **扩展单进程到多进程**   * **进程控制块 (PCB)**：为每个进程创建一个进程控制块，用于保存该进程的状态信息，如寄存器状态、程序计数器、优先级、内存地址等。操作系统通过PCB来区分和管理每个进程。 * **内存分配**：每个进程需要独立的地址空间，系统需要分配内存区域，并在上下文切换时切换到不同的内存空间，以确保进程间互不干扰。 * **进程状态管理**：进程有多种状态（如就绪、运行、等待、阻塞等），操作系统需要管理每个进程的状态转移，使得进程可以在不同状态间切换。 * **上下文切换**：当一个进程的时间片用完，或进入等待状态时，操作系统需要保存当前进程的状态，并加载下一个进程的状态，执行上下文切换。   **扩展中断以支持多进程**   * **时钟中断**：时钟中断是实现多进程调度的关键。操作系统会定期触发时钟中断，检查当前进程是否用完了分配的时间片。如果时间片到期，就进行进程切换，调度下一个进程执行。 * **中断处理程序**：在时钟中断中，操作系统会调用中断处理程序，保存当前进程的上下文信息（如寄存器值和程序计数器），然后执行调度算法，选择一个新的进程，并恢复该进程的上下文。 * **中断向量表**：中断向量表存储了各种中断的处理程序地址。当中断发生时，系统会跳转到中断处理程序执行相应的操作。在多进程系统中，时钟中断的处理程序需要支持进程的调度和切换。   书上的代码运行结果如下：    图中打印的字符“#”说明书上刚刚所增加的代码已经在正确运行。  书上主要是进程体的准备——》相关变量和宏——》进程表初始化——》LDT——》中断处理程序这几部分作了修改，实现了多进程  **2. 如何实现系统调用**  系统调用的实现步骤  系统调用入口：用户进程通过软中断（如 int 指令）或特殊指令（如 syscall）发起系统调用。这会触发CPU从用户态切换到内核态，调用内核的中断处理程序。  **传递参数：**用户进程在调用系统调用时，需要传递调用号和参数。这些参数通常通过寄存器或栈传递给内核。系统调用号用于在系统调用表中查找对应的内核服务函数。  **系统调用分派：**操作系统根据系统调用号查找系统调用表（sys\_call\_table），确定用户进程请求的操作。系统调用表是一个函数指针数组，包含了所有系统调用的入口地址。  **执行系统调用：**内核根据查找结果调用对应的系统调用服务函数，执行所需的操作（如文件读写、进程创建等）。  **返回结果：**系统调用完成后，返回结果（或错误代码）给用户进程，恢复用户态的寄存器和栈，恢复上下文信息。  **返回用户态：**系统调用结束后，操作系统返回到用户态继续执行用户进程  **系统调用表**：sys\_call\_table 是一个数组，存放每个系统调用的函数指针。  有如下定义：  void \*sys\_call\_table[] = {sys\_read, sys\_write, sys\_open, sys\_close, ...};  每个系统调用都有一个唯一的系统调用号，操作系统根据系统调用号在表中查找对应的函数。  **系统调用流程**：  用户态进程调用 int 0x80（假设这是系统调用的中断号）。  CPU切换到内核态，执行系统调用中断处理程序。  根据系统调用号找到并执行对应的内核函数。  函数执行完毕后，将返回值传回用户进程。  按照以上步骤，实验结果如图所示：    A后面紧跟的个数表达的是这个A之前出现的#个数，也就是时钟中断的个数，可以看到正常运行  **3.进程调度问题，弄清楚实现调度的基本思路**  **基本思路是决定哪个进程在何时可以使用CPU，参考书上讲的主要是优先级调度**  **（1） 进程调度时机**  进程调度发生在以下几种情况下：  **进程终止**：当前进程完成任务，退出CPU。  **进程进入等待**：进程需要等待I/O或其他资源，不再需要CPU。  **时间片用完**：分配给当前进程的时间片耗尽，需要将CPU分配给其他进程。  **中断发生**：例如时钟中断，操作系统检查是否需要切换进程  **（2） 调度算法**  调度算法决定了在进程调度时应该选择哪个进程执行。常见的调度算法包括：  **先来先服务调度算法 (FCFS)，短作业优先调度算法 (SJF)** ，**优先级调度算法，轮转调度算法 (RR)** ，**多级反馈队列调度算法**：为进程分配多个队列，根据运行时间和等待情况调整优先级，适用于多任务和交互环境  我们重点学习优先级调度和多级反馈队列调度  **（3） 进程切换流程**   * **保存当前进程状态**：将当前进程的寄存器、程序计数器、堆栈指针等信息保存到PCB（进程控制块）中，以便下次继续运行。 * **恢复下一个进程状态**：从下一个要执行的进程的PCB中恢复寄存器、程序计数器等信息，切换到该进程的上下文。 * **CPU控制权转移**：将CPU控制权转移到新的进程，开始执行新进程的代码   书上进程调度的运行实例如下：    可以看出打印出的字符的个数之比非常接近15:5:3，这主要是因为采用了优先级调度和延迟函数 | | | | | |
| 1. 实验过程分析   （详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等） | | | | | |
| 故障及其排除方法：  **（1）makefile文件错误**  还是要加上m32 -fno-stack-protector  同时每次make之前都不要忘记clean  **（2）调度失效**  调度器未按预期的时间片切换进程，导致某些进程长期占用CPU，有可能是时间片计数器未正确减少，或者被其他操作复位。调度器对时间片的控制逻辑错误，未能按时切换。  解决：  检查时间片的计数逻辑，确保每次时钟中断时，时间片计数器正常递减。  使用调试输出显示时间片计数值，检查其是否在预期范围内变化。  确保在时间片耗尽后，调度器切换到下一个进程 | | | | | |
| 1. 实验结果总结   （对实验结果进行分析，完成思考题目，并提出实验的改进意见） | | | | | |
| **结果分析**：  **1. 多进程的实现效果**  在实验中，我们成功地从单进程扩展到多进程系统。通过初始化多个进程控制块 (PCB)，并在时钟中断中实现了进程的切换，多个进程能够在系统中并行运行。实验结果表明，多进程的实现是正确的，每个进程在调度过程中能够顺利地保存和恢复状态，并在时间片到期后切换到下一个进程。这一结果验证了多进程管理机制的有效性，也证明了我们对PCB和进程状态管理的实现是正确的。  **2. 系统调用的功能验证**  实验中实现了基本的系统调用机制，例如 get\_ticks() 函数。用户进程可以通过系统调用访问内核服务，例如获取系统的时钟滴答数。实验结果表明，系统调用的参数传递和内核函数的执行均能顺利完成，用户进程成功获取了所需的信息。通过实验验证了系统调用的实现方法，表明我们在用户态和内核态之间的通信机制构建是正确的，系统调用接口也能够提供基本的服务功能。  **3. 进程调度算法的效果分析**  我们在实验中实现了优先级调度算法，并成功地调试运行了多个示例进程。结果打印输出无误，符合预期。结合思考题以及上面的实验过程可以看到更加详细的分析  **思考题目：**  **1.在单进程的基础上扩展实现多进程要考虑哪些问题？**   **进程控制块 (PCB)**：需要为每个进程创建并维护PCB，PCB中包含进程的状态、寄存器值、栈指针、优先级等信息。   **内存管理**：多进程需要独立的地址空间，系统需确保每个进程的内存相互隔离，避免干扰。   **中断处理**：多进程系统需要支持中断，以便在特定事件（如时间片到期）发生时进行进程切换。   **上下文切换**：当一个进程的执行被中断时，系统需要保存当前进程的上下文（寄存器值、程序计数器等），并切换到新的进程上下文。   **同步与互斥**：多进程系统需要机制来控制进程之间的同步和互斥，防止多个进程同时访问共享资源引发冲突。   **调度机制**：多进程系统需要设计合理的调度算法，根据优先级、等待时间等因素来决定哪个进程优先执行。  **2. 画出以下关键技术的流程图：– 初始化多进程控制块的过程、扩展初始化LDT和TSS**  **（1）初始化多进程控制块的过程**   1. 创建新的PCB结构，并分配一个进程ID。 2. 初始化PCB的基本信息，如状态、优先级、程序计数器、寄存器状态等。 3. 将PCB链接到进程队列或进程表中，以便操作系统管理。 4. 为进程分配独立的栈空间，初始化栈指针。 5. 设置进程的初始状态为“就绪”或“新建”。 6. **流程图**     **（2）扩展初始化LDT和TSS**   1. 为每个进程分配一个本地描述符表（LDT），以支持独立的进程地址空间。 2. 初始化LDT，设置各段的基址和限长。 3. 初始化任务状态段（TSS），存储进程切换时的寄存器值、栈指针等信息。 4. 将LDT和TSS的地址加入到全局描述符表（GDT）中，方便操作系统访问 5. **流程图：**   在这里插入图片描述  **3.如何修改时钟中断来支持多进程管理，画出新的流程图。**  **流程顺序如下：**  **（1）保存上下文**：在时钟中断发生时，首先保存当前进程的上下文信息（包括寄存器值和程序计数器）。  **（2）调度判断**：检查当前进程的时间片是否用完。如果用完，执行调度算法，选择下一个执行的进程。  **（3）更新时间片**：如果选择新的进程，将新进程的时间片重新加载到计数器中。  **（4）恢复上下文**：将新进程的上下文信息恢复到寄存器中。  **（5）返回**：从中断中返回，执行新进程  **流程图：**    **4. 系统调用的基本框架是如何的，应该包含哪些基本功能，画出流程图**  **基本框架：**   **系统调用入口**：用户态进程通过软件中断或特殊指令进入内核态，调用系统调用。   **参数传递**：系统调用号和参数通过寄存器或栈传递给内核。   **系统调用分派**：内核通过系统调用号查询系统调用表，找到对应的服务函数。   **执行系统调用**：调用对应的服务函数，执行用户进程请求的操作（如文件操作、进程控制等）。   **返回结果**：服务函数将执行结果传递回用户态，并恢复用户态的上下文。   **返回用户态**：系统调用完成后，返回用户态继续执行。  **流程图**  在这里插入图片描述  **5. 如何操控可编程计数器？**  查阅资料如下：  可编程计数器（通常用于时钟中断）通过特定的I/O端口控制，例如Intel 8253/8254计数器。主要步骤如下，按照先后顺序列出：  **设置模式**：通过写入控制寄存器，设置计数器的工作模式（如模式3：方波产生）。  **加载初始值**：将计数初始值写入计数器。这个值决定了计数器溢出并触发中断的时间。  **启动计数**：计数器开始从初始值递减，到达0时触发中断。  **读取/重设计数器**：可随时读取当前计数器的值，或重设初始值改变中断频率。  下面给出了一个简单地计数延迟示例    HZ：系统时钟每秒的滴答数（需要根据你的实际系统定义）。  milli\_delay(int milli\_sec)：延迟指定的毫秒数。  通过get\_ticks()记录当前滴答数，进入一个空循环，直到经过的时间（计算公式：(get\_ticks() - t) \* 1000 / HZ）满足指定的毫秒延迟milli\_sec    如图所示使用了延迟函数的进程，发生了很多次中断重入，打印出来的#变多了，这就是一个明显的效果  **6. 进程调度的框架是怎样的？优先级调度如何实现？**  **进程调度框架**   1. **就绪队列**：系统维护一个就绪队列，保存所有等待执行的进程。 2. **调度算法**：调度器根据算法选择一个进程作为下一个运行的进程。 3. **上下文切换**：如果当前进程需要切换，将当前进程的上下文保存到其PCB中，并加载新进程的上下文。 4. **时间片管理**：每个进程分配一定时间片，时间片用完后重新调度。   **优先级调度**   1. **优先级分配**：每个进程分配一个优先级，优先级越高，进程越先被调度。 2. **动态优先级**：某些实现中，进程的优先级会随等待时间或CPU时间的增加而调整。 3. **抢占机制**：如果一个高优先级进程进入就绪状态，则系统会中断当前进程，切换到高优先级进程。   实现可以参考下面动手做的多级反馈队列，是队列之间的优先级，与每个进程之间的优先级类似  **7. 动手做：修改例子程序的调度算法，模拟实现一个多级反馈队列调度算法，并用其尝试调度多个任务。注意，抢占问题，注意时间片问题。鼓励使用其他更复杂的调度算法，如CFS等**  **主要思路：**   **队列设置**：创建多个优先级队列，不同队列对应不同的时间片。   **进程调度**：从高优先级队列开始调度，优先执行高优先级队列的进程。若队列为空，则检查下一个优先级队列。   **时间片耗尽**：若进程的时间片耗尽且未完成任务，则降低其优先级，放入下一层的队列中。   **动态调整**：系统可以根据进程的等待时间和响应情况，将长时间等待的低优先级进程提升到高优先级队列中。   **抢占与切换**：当高优先级队列有新进程到达时，系统可以中断当前执行的低优先级进程，转而执行高优先级进程  **编程实现：使用C语言模拟内核工作过程（否则还要修改大量源代码文件，makefile也需要修改），代码讲解参见注释**  #include <stdio.h>  #include <stdlib.h>  #include <stdbool.h>  #include <unistd.h>  #define NUM\_QUEUES 3      // 队列数量  #define NUM\_PROCESSES 5   // 进程数量  typedef struct {      int pid;              // 进程ID      int burst\_time;       // 进程的总执行时间      int remaining\_time;   // 剩余的执行时间      int queue\_level;      // 当前所在的队列级别  } Process;  typedef struct {      Process\* queue[NUM\_PROCESSES];      int front;      int rear;      int time\_quantum;     // 队列的时间片  } Queue;  Queue queues[NUM\_QUEUES];  void init\_queues() {      for (int i = 0; i < NUM\_QUEUES; i++) {          queues[i].front = 0;          queues[i].rear = 0;          queues[i].time\_quantum = (i + 1) \* 2; // 时间片递增      }  }  bool is\_queue\_empty(Queue\* q) {      return q->front == q->rear;  }  void enqueue(Queue\* q, Process\* p) {      q->queue[q->rear++] = p;  }  Process\* dequeue(Queue\* q) {      return q->queue[q->front++];  }  void run\_process(Process\* p, int time\_quantum) {      int run\_time = (p->remaining\_time < time\_quantum) ? p->remaining\_time : time\_quantum;      printf("Running Process %d for %d units\n", p->pid, run\_time);      p->remaining\_time -= run\_time;      sleep(1); // 模拟进程执行  }  void mlfq\_scheduler(Process processes[]) {      init\_queues();      for (int i = 0; i < NUM\_PROCESSES; i++) {          processes[i].queue\_level = 0;          enqueue(&queues[0], &processes[i]);      }      while (true) {          bool all\_queues\_empty = true;          for (int level = 0; level < NUM\_QUEUES; level++) {              Queue\* q = &queues[level];              if (!is\_queue\_empty(q)) {                  all\_queues\_empty = false;                  Process\* p = dequeue(q);                  run\_process(p, q->time\_quantum);                  if (p->remaining\_time > 0) {                      if (level < NUM\_QUEUES - 1) {                          p->queue\_level++;                          enqueue(&queues[level + 1], p);                      } else {                          enqueue(q, p);                      }                  } else {                      printf("Process %d completed.\n", p->pid);                  }              }          }          if (all\_queues\_empty) break; // 所有队列为空，调度结束      }  }  int main() {      Process processes[NUM\_PROCESSES] = {          {1, 6, 6, 0},          {2, 8, 8, 0},          {3, 7, 7, 0},          {4, 3, 3, 0},          {5, 4, 4, 0},      };      mlfq\_scheduler(processes);      return 0;  }  运行结果如下：打印输出了正确的顺序    补充：在实验源码中添加代码实现该功能  修改schedule，给出部分代码，思路为寻找第一个不为-1的元素，找到后查看进程剩余时间片，如果大于0则进行调度，小于等于0则移到下一级队列中（通过数组元素移动）  image-20211118201821933  运行结果如下：  image-20211118202753608  发现A和B各自运行完5个时间片后，进入中优先级队列，A运行两个时间片后C到达进行抢占，C运行5个时间片后进入中优先级队列，A继续运行8个时间片，之后转入低优先级队列，B开始运行，然后C，循环往复  **8. 思考题：从用户态进程读和写内核段的数据，看能否成功**  我认为是**不允许**的，原因如下：  **（1）内存保护机制：**现代操作系统使用内存保护机制来区分用户态和内核态，防止用户态进程访问内核态的敏感数据。这是为了确保系统的安全性和稳定性。如果允许用户进程直接访问内核数据，恶意或有错误的程序可能会破坏系统，造成崩溃或安全漏洞。  **（2）权限控制：**内核态具有更高的权限，能够直接访问和控制硬件资源。用户态进程没有这样的权限，它只能通过系统调用的方式间接地请求内核执行某些操作。当用户进程尝试访问内核段数据时，CPU会检测到这种不合法的访问，并产生异常，通常会触发“段错误”或“访问违规”之类的错误。  **（3）系统调用的设计：**为了实现用户态与内核态的交互，操作系统提供了一套系统调用接口。这些系统调用使得用户进程可以在受控的环境中请求内核执行特定任务（如文件操作、进程管理等），而不直接操作内核数据。因此，通常情况下，用户态进程只能通过系统调用来间接读取或修改内核数据，且这些访问是受内核代码控制和保护的  **实际实验**    为了在用户态和内核态之间传递数据，通常会通过**系统调用**实现，从而在安全的环境下让内核执行数据的读取或写入操作，而不是直接访问内核数据  在这里插入图片描述  看来可以通过系统调用实现  **改进意见：**  **（1）引入更多调度算法的实现和比较**  **改进建议**：在多级反馈队列调度算法之外，添加其他调度算法（如最短作业优先调度算法SJF和完全公平调度算法CFS）的实现，并让学生比较不同算法的优缺点。  **目标**：更全面地了解不同调度算法的应用场景和性能差异，尤其是在不同工作负载下的表现  **（2）增加抢占式和非抢占式调度的对比**  **改进建议**：在实验中实现和对比抢占式调度和非抢占式调度的效果。  **目标**：帮助理解两者的区别，熟悉抢占式调度的实现细节和优先级管理  **（3）优化代码结构与可读性**  **改进建议**：提供更清晰的代码模板和注释规范，得到具有良好可读性的代码。这个orange上的代码的可读性很差，注释也不太理解，建议重做 | | | | | |
|  | | | | | |
| 1. 各人实验贡献与体会（每人各自撰写） | | | | | |
| 实际上每个人都各自独立地完成了实验  1.王亚鹏：全程参与实验，编写多级反馈队列算法及验证准确性，实验报告编写  **理解系统调用的实现框架**  在本次实验中，我们实现了简单的系统调用框架。通过设计系统调用并实现了get\_ticks()等基本功能，理解了用户态进程如何通过系统调用来访问内核功能  **调度算法的实现**  进程调度是本次实验的核心部分。通过实现多级反馈队列调度算法，体验到了不同调度算法在分配CPU资源时的策略差异和设计思路。多级反馈队列调度算法通过动态调整进程的优先级，使得短进程能够更快地得到响应，同时确保长进程不会被饿死  **调试过程**  在实验过程中，遇到了一些挑战。尤其是在实现调度算法时，如何正确保存和恢复进程状态、如何控制时间片的使用等都需要仔细分析和反复调试。通过不断调试，成功解决了问题  **系统调用安全性**  在实验中，我们探讨了用户态进程直接访问内核段数据的安全性问题。我了解到，为了保证系统的稳定性和安全性，用户进程必须通过系统调用间接访问内核数据，而不能直接操作内核内存   1. 杨依磊：全程参与实验   多进程实现与调度通过实验，我运行并理解了在操作系统中实现了多进程管理的程序。该程序在单进程的基础上，调整中断处理以支持多进程操作扩展实现了多进程。我理解了初始化多进程控制块的步骤，以及如何扩展LDT和TSS的初始化，使得多个进程能够独立运行。时钟中断的修改为支持多进程管理，需要对时钟中断进行修改。 我通过绘制新的流程图直观地理解了时钟中断的处理过程。修改后的时钟中断可以切换进程，从而实现了基本的进程切换机制。系统调用框架系统调用是用户程序与操作系统之间交互的重要桥梁。在本次实验中，我了解了系统调用如何包含各项基本功能，并绘制了相应的流程图。通过运行并调试程序，我理解了如何利用系统调用在用户态与内核态之间传递数据。多级反馈队列调度算法的实现实验中，我在调度算法的例子程序基础上进行了修改，尝试实现了多级反馈队列调度算法。此外，通过实验模拟多个任务的调度，深刻体会到时间片管理的重要性以及抢占调度的实现难点。  3.杜泓波：全程参与实验  在本次多进程实验中，我深入理解了操作系统中进程管理的核心原理，包括多进程的实现机制、系统调用的处理过程、以及进程调度的算法。实验的设计让我更加清晰地认识到操作系统如何通过进程控制块（PCB）来管理多个进程，并且如何利用中断机制进行上下文切换，确保系统能够高效、平稳地处理多个任务。  首先，通过学习多进程的原理，我理解了操作系统如何通过时间片轮转调度各个进程，并实现进程的动态切换。这一过程离不开中断机制，特别是时钟中断，能够触发调度器检查当前进程的状态，并决定是否进行进程切换。此外，我还深入了解了如何通过软中断实现系统调用，这一机制使得用户程序能够安全地请求操作系统服务。  在进程调度方面，实验中使用了多级反馈队列调度算法，它的自适应特性让我对调度的优化有了更深的体会。特别是如何根据进程的执行时间动态调整其优先级，从而提高系统效率。通过观察时间片的动态调整，我也理解了在多任务环境下，如何平衡各个进程的响应时间和执行时间。  最具挑战性的一部分是时钟中断的改进，通过修改时钟中断的处理函数，实现自动调度判断，确保系统能根据时钟中断周期性地切换进程。这个环节让我对中断处理和调度程序的协作有了更深的理解，同时也加深了对操作系统内核工作的认识。  总体来说，这次实验帮助我更好地理解了操作系统的多进程管理机制，尤其是在调度和中断处理上的应用。同时，通过实际的代码实现，我也能更直观地体会到操作系统内部复杂的交互和资源管理过程。   1. 侯名扬：全程参与实验   1、理解了多进程实现与调度：  多进程系统允许操作系统在同一时刻运行多个进程。每个进程在内存中拥有独立的地址空间和资源，系统通过时间片轮转的方式在各进程之间切换。多进程的核心原理在于操作系统内核通过进程控制块（PCB）来管理每个进程的状态，并提供中断和上下文切换的机制，以便在多个进程间进行调度和管理。  2、加深对多级反馈队列调度算法的运用  多级反馈队列调度算法是一种将进程根据优先级分层的调度方式。它使用多个队列，每个队列对应一个优先级，并且随着进程的运行和等待情况动态调整其所在的队列。  3、系统调用的功能验证  实验中实现了基本的系统调用机制，例如 get\_ticks() 函数。用户进程可以通过系统调用访问内核服务，例如获取系统的时钟滴答数。 | | | | | |
| 1. 教师评语   （实验报告的考评：依据实验内容完整度、实验步骤清晰度、实验结果与分析正确性、实验心得与思考的全面性、实验报告文档的规范性等五个维度综合考评）   |  |  | | --- | --- | | 85-100 | * 实验内容完整或者有超出课程实验大纲的内容； * 实验步骤详尽，能够体现完整的实验过程； * 实验结果正确且实验数据分析得当； * 实验心得与思考全面并且有自己的独立思考； * 实验报告文档规范、排版整齐。 | | 75-84 | * 实验内容较为完整； * 实验步骤较为详尽，能够体现实验过程； * 实验结果正确且实验数据分析较为得当； * 实验心得与思考全面； * 实验报告文档规范、排版较为整齐。 | | 60-74 | * 实验内容有缺失； * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | 60以下 | * 实验内容严重缺失、实验态度不够端正 * 实验步骤不够详尽，不能够体现完整的实验过程； * 实验结果部分正确； * 实验心得与思考无或者不够深入； * 实验报告文档规范性有待增强。 | | | | | | |
|  | | | | | |
| **教师评分（请填写好姓名、学号）** | | | | | |
| 姓名 | | 学号 | | 分数 | |
| 王亚鹏 | | 2022302181161 | |  | |
| 杨依磊 | | 2022302181159 | |  | |
| 杜泓波 | | 2022302181162 | |  | |
| 侯名扬 | | 2022302181165 | |  | |
| 教师签名：  年 月 日 | | | | | |