FDU 操作系统 1. CPU 的虚拟化

本文参考以下教材：

- Operating Systems: Three Easy Pieces (R. H. Arpaci-Dusseau & A. C. Arpaci-Dusseau) Chapter $4 \sim 10$
- 操作系统: 三个简单的部分 (王海鹏 译) 第 $4 \sim 10$ 章

欢迎批评指正!

1.1 抽象: 进程

通过让一个进程只运行一个时间片，然后切换到其他进程，操作系统提供了存在多个虚拟 CPU 的假象.
这就是**时分共享** (time sharing) CPU 技术，允许用户如愿运行多个并发进程.
潜在的开销就是性能损失，因为如果 CPU 必须共享，那么每个进程的运行就会慢一点.

要实现 CPU 的虚拟化，操作系统就需要一些低级机制以及一些高级策略.
机制 (mechanism) 是一些低级方法或协议，实现了所需的功能，例如上下文切换 (context switch).
在这些机制之上，操作系统中有一些**策略** (policy)，即做出某种决定的算法，例如调度策略 (scheduling policy)

1.1.1 进程

操作系统为正在运行的程序提供的抽象，就是所谓的**进程** (process)
进程的机器状态 (machine state):

- 进程可以访问的内存，称为地址空间 (address space)
- 寄存器，例如程序计数器和栈指针等等
- 访问 $\mathrm{I/O}$ 设备的信息，例如当前打开的文件列表

进程应当提供的 API:

- 创建 (create): 创建新进程的接口
- 销毁 (destroy): 强制销毁进程的接口
- 等待 (wait): 等待进程停止运行的接口
- 状态 (state): 获得进程状态信息的接口
- 其他控制 (miscellaneous control): 例如暂停进程和恢复进程的接口

1.1.2 进程创建

程序如何转化为进程:

- ① 将**代码**和**静态数据** (例如初始化变量) 加载到进程的地址空间中
  (现代操作系统惰性执行该过程，即仅在程序执行期间加载所需的代码或数据)
- ② 为程序的**运行时栈** (run-time stack) 分配内存，以存放局部变量、函数参数和返回地址
- ③ 为程序的**堆** (heap) 分配内存
  (堆用于显式请求的动态分配内存，程序用 $\mathrm{malloc}()$ 和 $\mathrm{free}()$ 来请求和释放这样的空间)
- ④ 其他初始化任务，特别是与 $\mathrm{I/O}$ 相关的任务

最后启动程序，从入口处 (即 $\mathrm{main}()$) 运行，将 CPU 的控制权转移给新创建的进程中，从而开始执行程序.

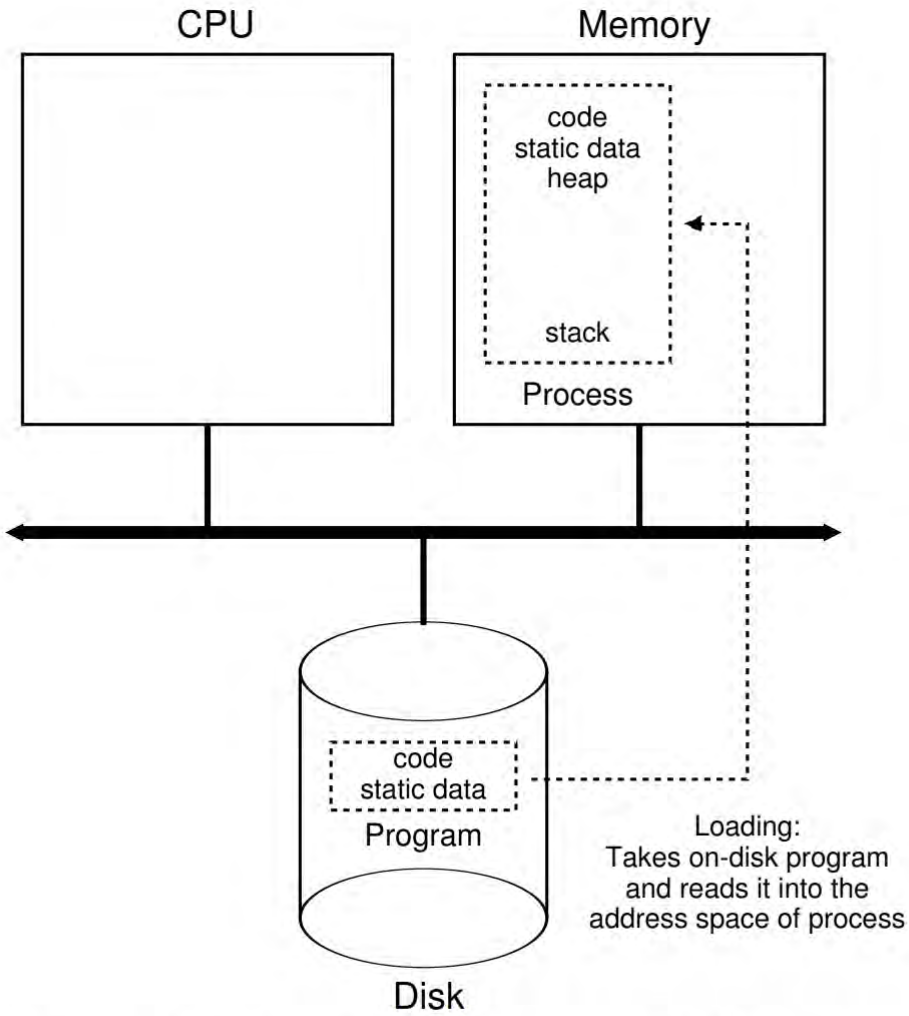


Figure 4.1: **Loading: From Program To Process**

1.1.3 进程状态

进程可以处于以下三种状态之一:

- 运行 (running): 进程正在 CPU 上运行
- 就绪 (ready): 进程已准备好运行，但由于某种原因，操作系统选择不在此时运行
- 阻塞 (blocked): 一个进程执行了某种操作，直到发生其他事件时才会准备运行

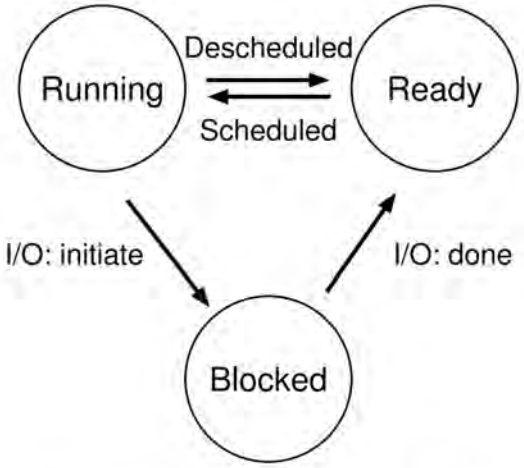


Figure 4.2: **Process: State Transitions**

从就绪到运行意味着该进程已经被**调度** (scheduled)
从运行转移到就绪意味着该进程已经**取消调度** (descheduled)
一旦进程被阻塞 (例如通过发起 $\mathrm{I/O}$ 操作)，OS 将保持进程的这种状态，
直到发生某种事件 (例如 $\mathrm{I/O}$ 完成)，进程再次转入就绪状态 (也可能立即再次运行，这取决于操作系统的决定).

为了跟踪每个进程的状态，操作系统可能会为所有就绪的进程保留某种**进程列表** (process list)，
以及跟踪当前正在运行的进程的一些附加信息.
操作系统还必须以某种方式跟踪被阻塞的进程.
当 $\mathrm{I/O}$ 事件完成时，操作系统应确保唤醒正确的进程，让它准备好再次运行.

除了运行、就绪和阻塞之外，还有其他一些进程可以处于的状态.
有时候系统会有一个初始 (initial) 状态，表示进程在创建时处于的状态.
另外，一个进程可以处于已退出但尚未清理的最终 (final) 状态 (在基于 UNIX 的系统中，这称为僵尸状态)
这个最终状态非常有用，因为它允许其他进程 (通常是创建进程的父进程) 检查进程的返回代码，
并查看刚刚完成的进程是否成功执行 (通常在基于 UNIX 的系统中，程序成功完成任务时返回零，否则返回非零).
完成后，父进程将进行最后一次调用 (例如 $\mathrm{wait}()$) 以等待子进程的完成，
并告诉操作系统它可以清理这个正在结束的进程的所有相关数据结构.

1.2 进程 API

本章将讨论 UNIX 系统中的进程创建.
UNIX 系统通过一对系统调用 $\mathrm{fork}()$ 和 $\mathrm{exec}()$ 来创建或终止进程.
它还通过第三个系统调用 $\mathrm{wait}()$ 来等待其创建的子进程执行完成.

Process Creation

Two ways to create a process

- Build a new empty process from scratch
- Copy an existing process and change it appropriately

Option 1: New process from scratch

- Steps
  - Load specified code and data into memory;
  - Create empty stack PCB: Process Control Block
  - Create and initialize PCB (make look like context-switch)
  - Put process on ready list
- Advantages: No wasted work
- Disadvantages: Difficult to setup process correctly and to express all possible options
  - Process permissions, where to write I/O, environment variables
  - Example: WindowsNT has call with 10 arguments

Option 2: Clone existing process and change

- Example: Unix fork() and exec()
  - Fork(): Clones calling process
  - Exec(char *file): Overlays file image on calling process
- fork()
  - Stop current process and save its state
  - Make copy of code, data, stack, and PCB
  - Add new PCB to ready list
  - Any changes needed to child process?
- exec(char *file)
  - Replace current data and code segments with those in specified file
- Advantages: Flexible, clean, simple
- Disadvantages: Wasteful to perform copy and then overwrite of memory

1.2.1 fork()

系统调用 fork() 用于创建新进程.
考虑以下程序 p1.c:

```c
#include <stdio.h>    // Standard I/O library for printf and fprintf
#include <stdlib.h>   // Standard library for exit function
#include <unistd.h>   // Unix standard library for fork and getpid

int main(int argc, char *argv[]) {
    // Print a message from the parent process (before fork), displaying its PID
    printf("hello world (pid:%d)\n", (int) getpid());

    // Fork a new process: creates a copy of the current process
    int rc = fork();

    // Check if fork() failed (returns negative value if it fails)
    if (rc < 0) {
        // Print error message to stderr and exit with a non-zero status
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    // If fork() succeeds, the new process is called the child process
    // fork() returns 0 in the child process
    else if (rc == 0) {
        // This block is executed by the child process
```

```
        // The child prints its own PID (different from the parent's PID)
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    }
    // fork() returns the PID of the child process to the parent
    else {
        // This block is executed by the parent process
        // The parent prints the PID of the child and its own PID
        printf("hello, I am parent of %d (pid:%d)\n", rc, (int) getpid());
    }

    // Return 0 to indicate successful execution
    return 0;
}
```

输出结果:

```
Linux:~/Desktop/OSTEP$ gcc -o p1 p1.c -Wall
Linux:~/Desktop/OSTEP$ ./p1
hello world (pid:3205)
hello, I am parent of 3206 (pid:3205)
hello, I am child (pid:3206)
Linux:~/Desktop/OSTEP$
```

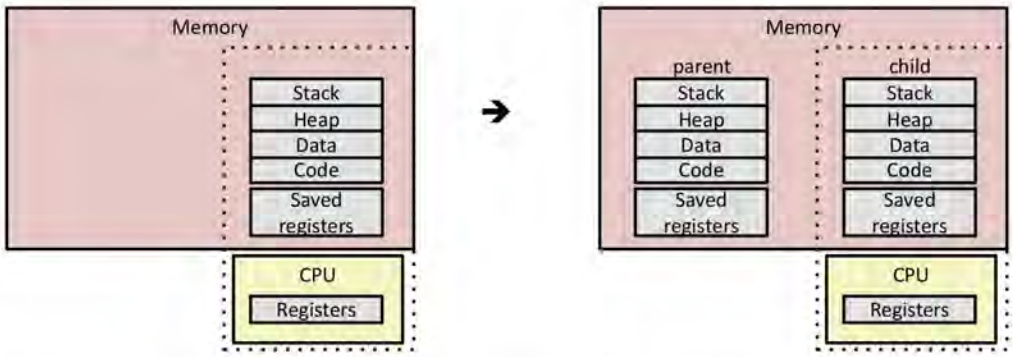
当上述程序 p1.c 刚开始运行时，进程输出一条 `hello world` 信息，以及其进程描述符 (process identifier, PID)
该进程的 PID 是 3205
在 UNIX 系统中，如果要操作某个进程 (例如终止进程)，就要通过 PID 来指明.

接着有趣的事情发生了:
进程调用了 fork() 系统调用，这是操作系统提供的创建新进程的方法.
新创建的进程几乎与调用进程完全一样
对操作系统来说似乎两个完全一样的 p1 程序在运行，并都从 fork() 系统调用中返回.
新创建的进程称为**子进程** (child)， 原来的进程称为**父进程** (parent).

子进程不会从 main() 函数开始执行 (因此总的来说 `hello world` 信息只输出一次)，
而是直接从 fork() 系统调用返回，就好像是它自己调用了 fork()
因此子进程并不是完全拷贝了父进程
具体来说，虽然它拥有自己的地址空间、寄存器、程序计数器等，但是它从 fork() 返回的值是不同的.
父进程获得的返回值是新创建子进程的 PID，而子进程获得的返回值是 0
我们的代码正是利用这个不同来让父进程和子进程输出不同的信息.

值得注意的是，上述程序的输出是不确定的.
简单起见，假设在单 CPU 的系统上运行 p1.c，那么当子进程被创建以后，它与父进程都有可能先于对方运行.
在之前的输出结果中，父进程先运行并输出信息
而在其他情况下，子进程可能先运行 (尽管我在 2 核机器上没有产生过这样的输出)

Conceptual View of `fork`



- Make complete copy of execution state
  - Designate one as parent and one as child
  - Resume execution of parent or child
- *Parent process* creates a new running *child process* by calling `fork`
- `int fork(void)`
  - Returns 0 to the child process, child's PID to parent process
  - Child is *almost* identical to parent:
    - Child get an identical (but separate) copy of the parent's virtual address space.
    - Child gets identical copies of the parent's open file descriptors
    - Child has a different PID than the parent
- `fork` is interesting (and often confusing) because it is called *once* but returns *twice*

1.2.2 wait()

有时候父进程需要等待子进程执行完毕，
这项任务由 wait() 系统调用 (或更完整的兄弟接口 waitpid()) 完成.
考虑以下程序 p2.c:

```
#include <stdio.h>   // Required for printf and fprintf functions
#include <stdlib.h>  // Required for exit function
#include <unistd.h>  // Required for fork and getpid functions
#include <sys/wait.h> // Required for wait function

int main(int argc, char *argv[]) {
    // Print the initial message with the process ID of the current process
(parent)
    printf("hello world (pid:%d)\n", (int) getpid());

    // Fork the process; `rc` will be 0 in the child process, and the child's PID
in the parent
    int rc = fork();

    // Error handling: if `fork()` returns a negative value, it means the fork
failed
    if (rc < 0)
    {
```

```
        // Print an error message to stderr and exit with status 1 to indicate
failure
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0) // Child process block
    {
        // In the child process: print the message and the child's process ID
        printf("hello, I am child (pid:%d)\n", (int) getpid());
    }
    else  // Parent process block
    {
        // The parent waits for the child process to finish using wait()
        int rc_wait = wait(NULL);

        // Print the message with the child process ID (rc), return value of
wait (rc_wait),
        // and the parent's process ID (getpid())
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n", rc, rc_wait,
(int) getpid());
    }

    // Return 0 to indicate the successful completion of the program
    return 0;
}
```

输出结果:

```
Linux:~/Desktop/OSTEP$ gcc -o p2 p2.c -Wall
Linux:~/Desktop/OSTEP$ ./p2
hello world (pid:7313)
hello, I am child (pid:7322)
hello, I am parent of 7322 (rc_wait:7322) (pid:7313)
Linux:~/Desktop/OSTEP$
```

在 p2.c 的例子中，子进程总是先输出结果.
好吧，它可能只是碰巧先运行，像以前一样，因此先于父进程输出结果.
但是如果父进程碰巧先运行，那么它会调用 wait()，延迟自己的执行，直到子进程执行完毕，wait() 才返回父进程.
(有些情况下, wait() 会在子进程退出之前返回，请阅读 man 手册获取更多细节)

Process Termination

- By invoking exit(0), the process notifies OS kernel to terminate.
- When to recycle resources inside kernel?
  - Just like when you graduate from the university.
  - When the parent call wait() to wait for the child's termination, it can reap all the resources corresponding to the child
  - What if the parent never wait?
  - The child becomes zombie, and the init process (pid = 1) will reap all.

```
linux> ./forks 7 &
[1] 6639
Running Parent, PID = 6639
Terminating Child, PID = 6640
linux> ps
  PID TTY          TIME CMD
 6639 ttyp9    00:00:03 forks
 6640 ttyp9    00:00:00 forks <defunct>
 6641 ttyp9    00:00:00 ps
```

`wait`: Synchronizing with Children

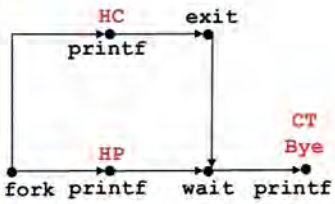
- **Parent reaps a child by calling the `wait` function**
- `int wait(int *child_status)`
  - Suspends current process until one of its children terminates
  - Return value is the **`pid`** of the child process that terminated
  - If **`child_status != NULL`**, then the integer it points to will be set to a value that indicates reason the child terminated and the exit status:
    - Checked using macros defined in `wait.h`
      - `WIFEXITED, WEXITSTATUS, WIFSIGNALED, WTERMSIG, WIFSTOPPED, WSTOPSIG, WIFCONTINUED`
      - See textbook for details

Common Usage of wait()

```
void fork9() {
    int child_status;

    if (fork() == 0) {
        printf("HC: hello from child\n");
        exit(0);
    } else {
        printf("HP: hello from parent\n");
        wait(&child_status);
        printf("CT: child has terminated\n");
    }
    printf("Bye\n");
}
```

forks.c



Feasible output(s):

HC	HP
HP	HC
CT	CT
Bye	Bye

Infeasible output:

```
HP
CT
Bye
HC
```

1.2.3 exec()

最后是 exec() 系统调用，它也是创建进程 API 的一个重要部分.
(事实上，它有几种变体: execl(), execle(), execlp(), execv(), execvp(), 请阅读 man 手册获取更多细节)
这个系统调用可以让子进程执行与父进程不同的程序.
考虑以下程序 p3.c:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <sys/wait.h>

int main(int argc, char *argv[])
{
    printf("hello world (pid:%d)\n", (int) getpid());

    int rc = fork();  // Create a new process
    if (rc < 0)
    {  // fork failed; exit
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)
    {  // child process
        printf("hello, I am child (pid:%d)\n", (int) getpid());

        char *myargs[3];
        myargs[0] = strdup("wc");        // program: "wc" (word count) which is
a shell built-in command
        myargs[1] = strdup("p3.c");      // argument: file to count words in
        myargs[2] = NULL;                // marks the end of the array (required
by execvp)

        execvp(myargs[0], myargs);  // runs the word count program on file p3.c

        // This line will only run if execvp fails
        printf("this shouldn't print out\n");

        // Free the dynamically allocated memory in case execvp fails
        free(myargs[0]);
        free(myargs[1]);
    }
    else
    {  // parent process
        int rc_wait = wait(NULL);  // Wait for the child process to finish
        printf("hello, I am parent of %d (rc_wait:%d) (pid:%d)\n",
               rc, rc_wait, (int) getpid());
    }

    return 0;
}
```

输出结果:

```
Linux:~/Desktop/OSTEP$ gcc -o p3 p3.c -Wall
Linux:~/Desktop/OSTEP$ ./p3
hello world (pid:5207)
hello, I am child (pid:5213)
  42  169 1314 p3.c
hello, I am parent of 5213 (rc_wait:5213) (pid:5207)
Linux:~/Desktop/OSTEP$
```

在这个例子中，子进程调用 execvp() 来针对源代码文件 p3.c 运行字符计数程序 wc
从而告诉我们该文件有多少行、多少单词，以及多少字节.

给定可执行程序的名称 (如 wc) 及需要的参数 (如 p3.c) 后，
exec() 会从可执行程序中加载代码和静态数据，并用它覆写自己的代码段 (以及静态数据)
堆、栈及其他内存空间也会被重新初始化.
然后操作系统执行该程序，将参数通过 argv 传递给该进程.
因此它并没有创建新进程，而是直接将当前运行的程序 (以前的 p3) 替换为不同的运行程序 (wc)
子进程执行 exec() 之后，几乎就像 p3.c 从未运行过一样.
对 exec() 的成功调用永远不会返回.

1.2.4 为什么这样设计

为什么要设计如此奇怪的接口，来完成简单的创建新进程的任务?
事实证明，这种分离 fork() 及 exec() 的做法在构建 UNIX shell 的时候非常有用
因为这给了 shell 在 fork 之后 exec 之前运行代码的机会
这些代码可以在运行新程序前改变环境，从而让一系列有趣的功能很容易实现.

shell 也是一个用户程序 (例如 tesh, bash 和 zsh 等等, 请选择一个并阅读它的 man 手册)
它首先显示一个提示符 (prompt)，然后等待用户输入.
你可以向它输入一个命令 (一个可执行程序的名称及需要的参数)
大多数情况下，shell 可以在文件系统中找到这个可执行程序
调用 fork() 创建新进程，并调用 exec() 的某个变体来执行这个可执行程序，调用 wait() 等待该命令完成.
子进程执行结束后，shell 从 wait() 返回并再次输出一个提示符，等待用户输入下一条命令.

考虑示例:

```
Linux:~/Desktop/OSTEP$ wc p3.c > word_count_p3.txt
```

运行之后我们在当前目录得到一个文件 word_count_p3.txt，其内容为:

```
  42  169 1314 p3.c
```

在上面的例子中，wc 的输出结果被重定向 (redirect) 到文件 word_count_p3.txt 中
shell 实现结果重定向的方式也很简单:
在完成子进程的创建后，shell 在调用 exec() 之前先关闭了标准输出 (standard output)，并打开文件 word_count_p3.txt
这样，即将运行的程序 wc 的输出结果就被发送到该文件，而不是打印在屏幕上.

考虑以下代码 p4.c:

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <fcntl.h>
#include <sys/wait.h>
#include <sys/stat.h>  // Include this header for S_IRWXU

int main(int argc, char *argv[])
{
    int rc = fork();  // Create a child process

    if (rc < 0)  // fork failed
    {
        // Print an error message if fork fails
        fprintf(stderr, "fork failed\n");
        exit(1);
    }
    else if (rc == 0)  // Child process
    {
        // Close the standard output file descriptor (STDOUT)
        close(STDOUT_FILENO);

        // Open (or create) a file for writing, and redirect stdout to this file
        open("./p4.output", O_CREAT | O_WRONLY | O_TRUNC, S_IRWXU);
        // O_CREAT: Create the file if it doesn't exist
        // O_WRONLY: Open the file for writing only
        // O_TRUNC: Truncate the file to 0 if it already exists
        // S_IRWXU: Give read, write, and execute permissions to the owner

        // Prepare arguments for execvp
        char *myargs[3];
        myargs[0] = strdup("wc");    // program: "wc" (word count)
        myargs[1] = strdup("p4.c");  // arg: file to count (p4.c in this case)
        myargs[2] = NULL;            // mark end of array

        // Replace the current process image with a new process image (wc
program)
        execvp(myargs[0], myargs);   // runs word count
```

```
        // If execvp fails, this line will execute
        fprintf(stderr, "execvp failed\n");

        // Free the dynamically allocated memory
        free(myargs[0]);
        free(myargs[1]);
    }
    else // Parent process
    {
        // Wait for the child process to finish
        int rc_wait = wait(NULL);
    }

    return 0;  // End of program
}
```

输出结果:

```
Linux:~/Desktop/OSTEP$ gcc -o p4 p4.c -Wall
Linux:~/Desktop/OSTEP$ ./p4
Linux:~/Desktop/OSTEP$
```

运行代码后得到了 p4.output 文件，其内容为:

```
  53  233 1730 p4.c
```

关于这个输出有两个有趣的地方:

- 首先，当运行 p4 程序后，好像什么也没有发生.
  shell 只是打印了命令提示符，等待用户的下一个命令.
  但事实并非如此，p4 确实调用了 fork 来创建新的子进程，之后调用 execvp() 来执行 wc
  屏幕上没有看到输出， 是由于结果被重定向到文件 p4.output
- 其次，当用 cat 命令打印输出文件 p4.output 时，能看到运行 wc 的所有预期输出:

  ```
  Linux:~/Desktop/OSTEP$ cat p4.output
    53  233 1730 p4.c
  Linux:~/Desktop/OSTEP$
  ```

UNIX 管道也是用类似的方式实现的，但用的是 pipe() 系统调用.
在这种情况下，一个进程的输出被链接到了一个内核管道 (pipe) 上，另一个进程的输入也被连接到了同一个管道上.
因此前一个进程的输出无缝地作为后一个进程的输入，许多命令可以用这种方式串联在一起，共同完成某项任务.
例如通过将 grep 和 wc 命令用管道连接在一起，就可以完成从一个文件中查找某个词，并统计其出现次数的功能:

```
Linux:~/Desktop/OSTEP$ grep -o main p4.c | wc -l
1
Linux:~/Desktop/OSTEP$ grep -o fork p4.c | wc -l
4
Linux:~/Desktop/OSTEP$
```

例如在 $\mathrm{p4.c}$ 文件中分别统计单词 "main" 和 "fork" 出现的次数，得到的结果分别是 1 和 4

1.3 受限直接执行

为了使程序尽可能快地运行，操作系统开发人员想出了一种技术——**受限的直接执行** (limited direct execution)
这个概念的 **"直接执行"** 部分很简单: 只需直接在 CPU 上运行程序即可.
因此当 OS 希望启动程序运行时，它会在进程列表中为其创建一个进程条目，为其分配一些内存，
将程序代码 (从磁盘) 加载到内存中，找到入口点 ($\mathrm{main}()$ 函数或类似的)，跳转到那里，并开始运行用户的代码.

下表展示了这种基本的直接执行协议 (没有任何限制)，使用正常的调用并返回跳转到程序的 $\mathrm{main}()$，并在稍后回到内核:

表 6.1　　直接运行协议（无限制）

操作系统	程序
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argc/argv 设置程序栈	
清除寄存器 执行 call main() 方法	
	执行 main() 从 main 中执行 return
释放进程的内存将进程 从进程列表中清除	

但 "直接执行" 会产生一些问题:

- 第一个问题:
  如果我们只运行一个程序，操作系统怎么能确保程序不做任何我们不希望它做的事 (例如 `while(1);`)?
- 第二个问题:
  当我们运行一个进程时，操作系统如何让它停下来并切换到另一个进程，从而实现虚拟化 CPU 所需的时分共享?

1.3.1 受限的操作

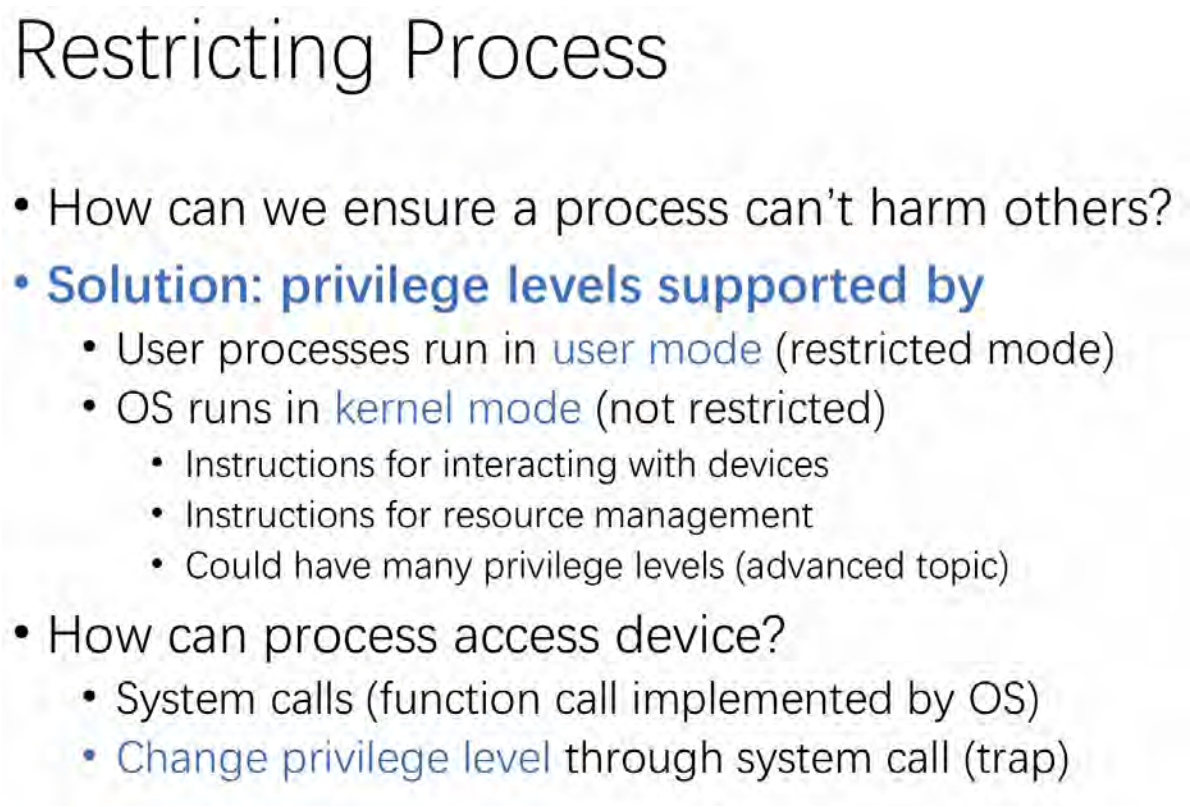
对于第一个问题:
如果对运行程序没有限制，那么操作系统将无法控制任何事情，因此会仅仅是一个 "库"
这对于任何一个有抱负的操作系统来说都是一件令人悲伤的事.

如果进程希望执行某种受限操作 (如向磁盘发出 $\mathrm{I/O}$ 请求或获得更多系统资源 (如内存))，该怎么办?
为此我们采用的方法是引入一种新的处理器模式，称为**用户模式** (user mode)
在用户模式下运行的代码会受到限制.
例如在用户模式下运行时，进程不能发出 $\mathrm{I/O}$ 请求.
这样做会导致处理器引发异常，操作系统可能会终止进程.

与用户模式不同的**内核模式** (kernel mode)，操作系统 (或内核) 就以这种模式运行.
在此模式下，运行的代码可以做它想做的事，包括特权操作 (例如发出 $\mathrm{I/O}$ 请求和执行所有类型的受限指令)

如果用户希望执行某种特权操作， 应该怎么做?
为了实现这一点，几乎所有的现代硬件都提供了用户程序执行**系统调用** (system call) 的能力.
系统调用允许内核小心地向用户程序提供某些关键功能
例如访问文件系统、创建和销毁进程、与其他进程通信，以及分配更多内存.

为执行系统调用，程序必须执行特殊的**陷阱指令** (trap instruction)
该指令同时跳入内核并将特权级别提升到内核模式.
一旦进入内核，系统就可以执行任何需要的特权操作 (如果允许)，从而为调用进程执行所需的工作.
完成后，操作系统调用一个特殊的**陷阱返回指令** (return-from-trap instruction)
该指令返回到发起调用的用户程序中，同时将特权级别降低，回到用户模式.



还有一个重要的细节没讨论:
陷阱如何知道在 OS 内运行哪些代码?
显然，发起调用的过程不能指定要跳转到的地址 (像进行过程调用时那样)
这样做让程序可以跳转到内核中的任意位置，这无疑是一个糟糕的主意.
因此内核必须谨慎地控制在陷阱上执行的代码，这通过在启动时设置**陷阱表** (trap table) 来实现.

当机器启动时，它在内核模式下执行，因此可以根据需要自由配置机器硬件.
操作系统做的第一件事，就是告诉硬件在发生某些异常事件时要运行哪些代码，通知硬件这些陷阱处理程序的位置.
在下次重新启动之前，硬件都会记住这些处理程序的位置，
并且知道在发生系统调用和其他异常事件时要做什么 (即跳转到哪段代码)

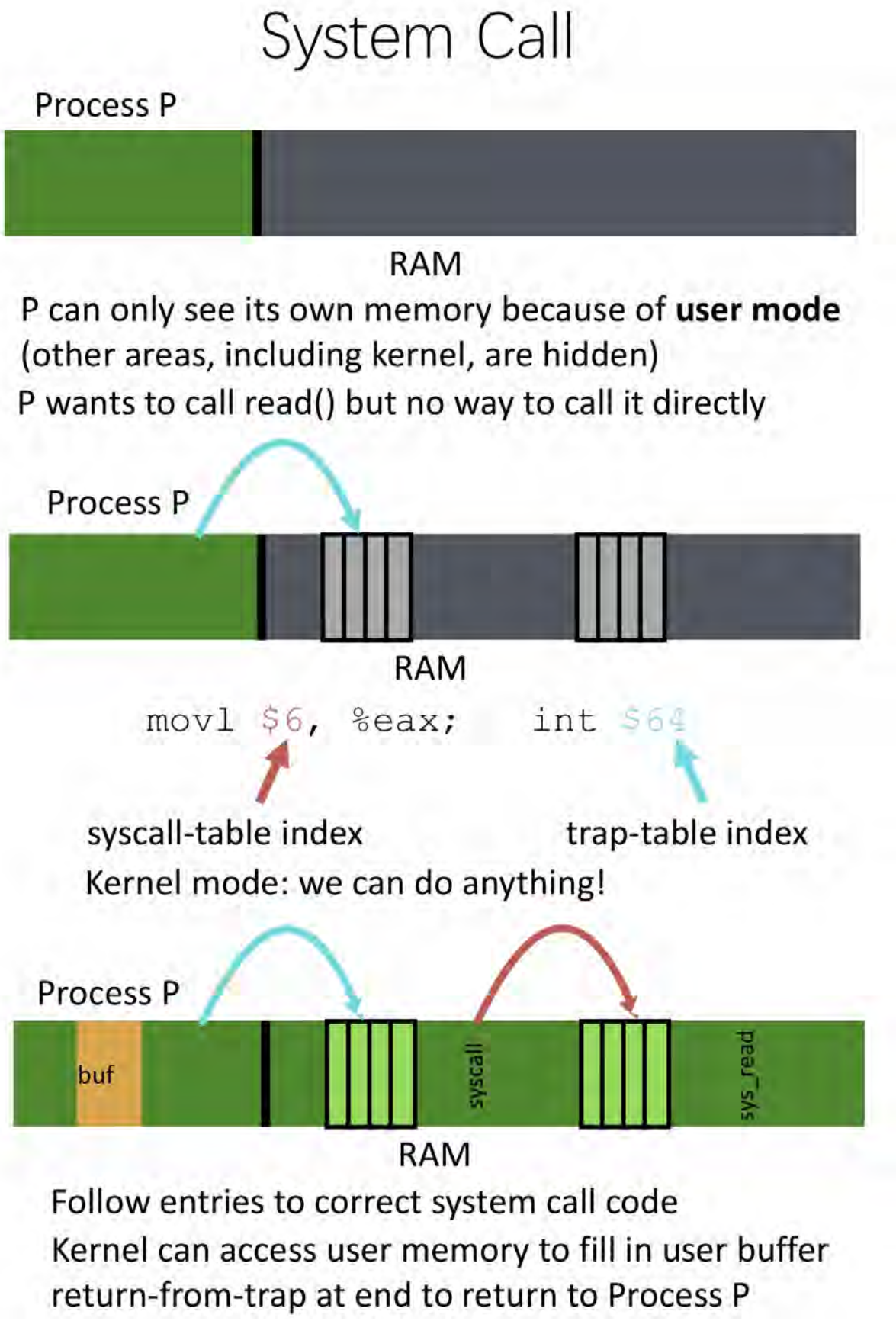
下表总结了**受限直接执行** (LDE) 协议.
其中我们假设每个进程都有一个内核栈
且在进入内核和离开内核时，寄存器 (包括通用寄存器和程序计数器) 分别被保存和恢复.

表 6.2 受限直接运行协议

操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住系统调用处理程序的地址	
操作系统@运行（内核模式）	硬件	程序（应用模式）
在进程列表上创建条目 为程序分配内存 将程序加载到内存中 根据 argv 设置程序栈 用寄存器/程序计数器填充内核栈 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到 main	
		运行 main …… 调用系统调用 陷入操作系统
	将寄存器保存到内核栈 转向内核模式 跳到陷阱处理程序	
处理陷阱 做系统调用的工作 从陷阱返回		
	从内核栈恢复寄存器 转向用户模式 跳到陷阱之后的程序计数器	
		……从 main 返回 陷入（通过 exit()）
释放进程的内存将进程 从进程列表中清除		

LDE 协议有两个阶段:

- 第一个阶段 (在系统引导时)，内核初始化陷阱表，并且 CPU 记住它的位置以供随后使用.
  内核通过特权指令来执行此操作 (所有特权指令均以粗体突出显示)
- 第二个阶段 (运行进程时)，在使用从陷阱返回指令开始执行进程之前，内核设置了一些内容
  (例如在进程列表中分配一个节点，分配内存)
  这会将 CPU 切换到用户模式并开始运行该进程.
  当进程希望发出系统调用时，它会重新陷入操作系统，然后再次通过从陷阱返回，将控制权还给进程.
  该进程然后完成它的工作，并从 $\mathrm{main}()$ 返回
  这通常会返回到一些存根代码，它将正确退出该程序 (例如通过调用 $\mathrm{exit}()$ 系统调用，这又将陷入 OS 中)
  此时 OS 清理干净，任务完成了.



1.3.2 进程切换

对于第二个问题:
操作系统如何重新获得 CPU 的控制权，以便它可以在进程之间切换?

(1) 重新获得控制权

① 协作方式: 等待系统调用

早期的某些操作系统采用的是**协作方式** (cooperative approach)
在这种风格下，操作系统相信系统的进程会合理运行.
运行时间过长的进程被假定会定期放弃 CPU，以便操作系统可以转而运行其他任务.
像这样的系统通常包括一个显式的 yield 系统调用
它什么都不干，只是将控制权交给操作系统，以便系统可以运行其他进程.

这种被动的方式并不太理想.
如果某个进程 (无论是恶意的还是充满缺陷的) 进入无限循环，并且从不进行系统调用
那么操作系统将无法重新获得 CPU 的控制权.
事实上在协作方式中，当进程陷入无限循环时，唯一的解决办法就是重启计算机.

---

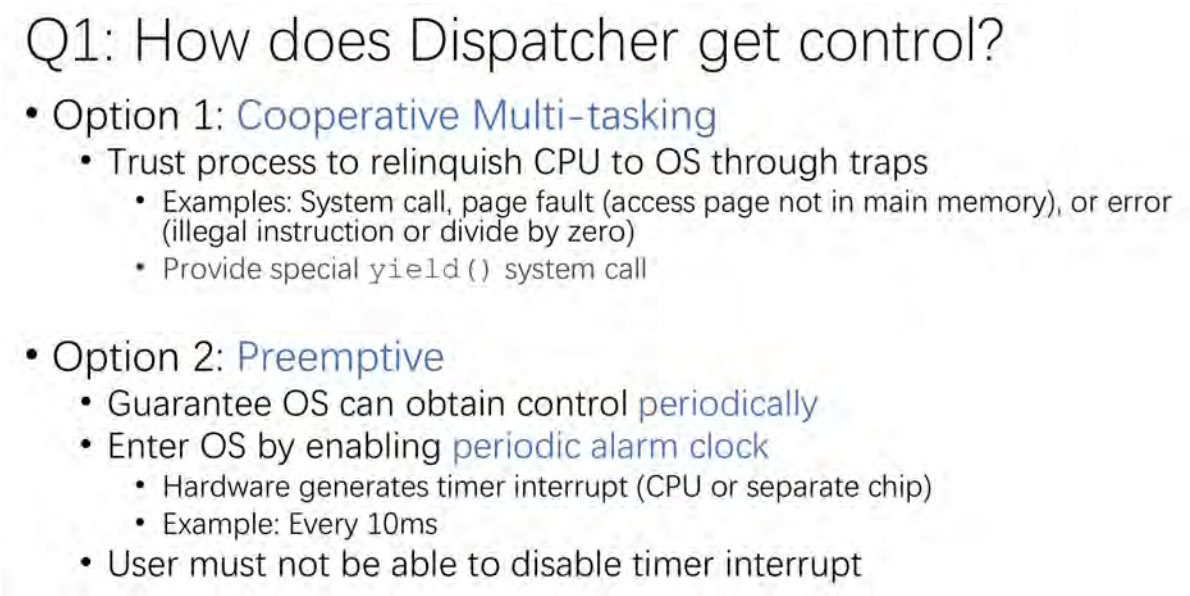
② 非协作方式: 操作系统进行控制

当进程不协作或陷入无限循环时，操作系统该如何获得 CPU 的控制权?
答案很简单: **时钟中断** (timer interrupt)
此硬件功能对于帮助操作系统维持机器的控制权至关重要.

时钟设备可以编程为每隔几毫秒产生一次中断.
产生中断时，当前正在运行的进程停止，操作系统中预先配置的**中断处理程序** (interrupt handler) 会运行.
此时操作系统重新获得 CPU 的控制权，因此可以做任何它想做的事 (例如停止当前进程，转而运行另一个进程)

- 首先，与系统调用类似地，在启动计算机的过程中，操作系统必须通知硬件哪些代码应在发生时钟中断时运行.
- 其次，在启动过程中，操作系统也必须启动时钟，这当然是一项特权操作.
  一旦时钟开始运行，操作系统就感到安全了，因为控制权每隔一段时间都会被归还给它.
  此时操作系统便可以自由运行用户程序.
  时钟也可以关闭 (这也是特权操作)，我们会有关并发的内容中详细讨论.

请注意，硬件在发生中断时有一定的责任，它们要为正在运行的程序保存足够的状态 (各种寄存器被保存入内核栈)
以便随后从陷阱返回指令能够正确恢复正在运行的程序，这与显式系统调用陷入内核时的行为非常相似.



(2) 保护和恢复上下文

既然操作系统已经重新获得了控制权，无论是通过系统调用协作，还是通过时钟中断强制执行，都必须决定:
是继续运行当前正在运行的进程，还是切换到另一个进程.
这个决定是由**调度程序** (scheduler) 做出的，我们把这部分内容放在后面介绍.

如果调度程序决定进行切换到另一个进程，操作系统就会执行一些底层代码，即**上下文切换** (context switch)
操作系统要做的就是为当前正在执行的进程保存一些寄存器的值 (到它的内核栈)
并为即将执行的进程恢复一些寄存器的值 (从它的内核栈)
这样一来，操作系统就可以确保最后执行从陷阱返回指令时，不是返回到之前运行的进程， 而是继续执行另一个进程.

为了保存当前正在运行的进程的上下文，操作系统会执行一些底层汇编代码，
来保存通用寄存器、程序计数器，以及当前正在运行的进程的内核栈指针，
然后恢复寄存器、程序计数器，并切换内核栈，供即将运行的进程使用.

通过切换栈，内核在进入切换代码调用时，是一个进程 (被中断的进程) 的上下文
在返回时，是另一进程 (即将执行的进程) 的上下文.
当操作系统最终执行从陷阱返回指令时，即将执行的进程变成了当前运行的进程.
至此上下文切换完成.

(3) 示例

假设进程 A 正在运行，然后被中断时钟中断.
硬件保存它的寄存器 (在内核栈中)，并进入内核 (切换到内核模式)

在时钟中断处理程序中，操作系统通过调度程序决定从正在运行的进程 A 切换到进程 B
此时它调用 $\mathrm{switch}()$ 例程，该例程仔细保存当前寄存器的值 (保存到 A 的进程结构)，
恢复寄存器进程 B (从它的进程结构)，然后切换上下文.
具体来说是通过改变栈指针来使用 B 的内核栈 (而不是 A 的)

最后操作系统从陷阱返回，恢复 B 的寄存器并开始运行它.

下表展示了整个进程切换过程的时间线:

表 6.3　　受限直接执行协议（时钟中断）

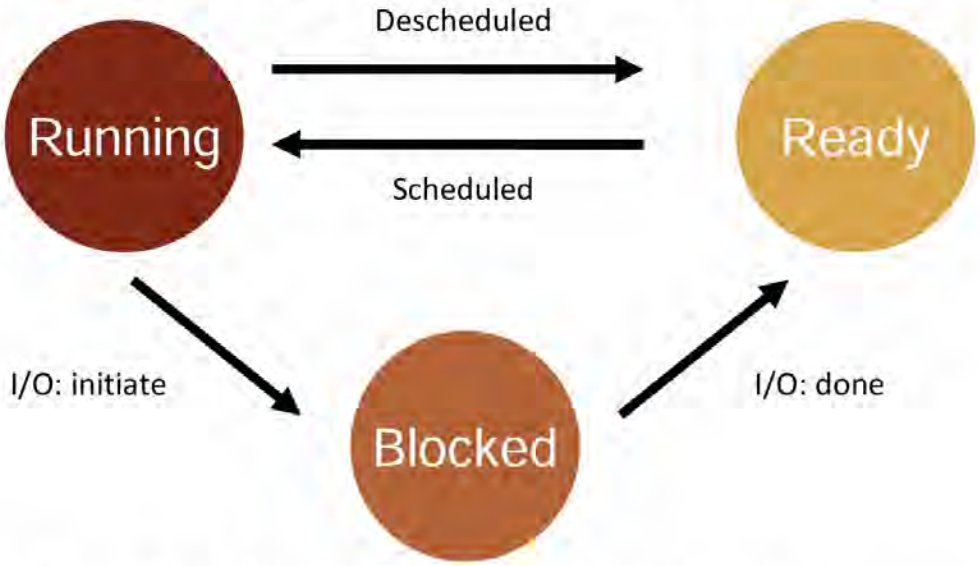
操作系统@启动（内核模式）	硬件	
初始化陷阱表		
	记住以下地址： 　系统调用处理程序 　时钟处理程序	
启动中断时钟		
	启动时钟 每隔 x ms 中断 CPU	
操作系统@运行（内核模式）	硬件	程序（应用模式）
		进程 A……
	时钟中断 将寄存器（A）保存到内核栈（A） 转向内核模式 跳到陷阱处理程序	
处理陷阱 调用 switch()例程 　将寄存器（A）保存到进程结构（A） 　将进程结构（B）恢复到寄存器（B） 从陷阱返回（进入 B）		
	从内核栈（B）恢复寄存器（B） 转向用户模式 跳到 B 的程序计数器	
		进程 B……

请注意，在此协议中，有两种类型的寄存器保存/恢复.

- 第一种是发生时钟中断的时候，运行进程的用户寄存器由硬件隐式保存，使用该进程的内核栈.
- 第二种是当操作系统决定从进程 A 切换到进程 B 时，内核寄存器被操作系统明确地保存，
  但这次被存储在该进程的进程结构的内存中.
  这个操作让系统从好像刚刚由进程 A 陷入内核， 变成好像刚刚由进程 B 陷入内核.

Problem 3: Slow Ops such as I/O?

- When running process performs a op that does not use CPU, OS switches to process that needs CPU (policy issues)
- OS must track mode of each process:
  - Running:
    - On the CPU (only one on a uniprocessor)
  - Ready:
    - Waiting for the CPU
  - Blocked:
    - Asleep: Waiting for I/O or synchronization to complete



- OS must track every process in system
  - Each process identified by unique Process ID (PID)
- OS maintains queues of all processes
  - Ready queue: Contains all ready processes
  - Event queue: One logical queue per event
    - e.g., disk I/O and locks
    - Contains all processes waiting for that event to complete

1.4 进程调度

1.4.1 简介

现在我们已经了解了运行进程的底层机制 (例如上下文切换)
接下来我们会介绍一系列的**调度策略** (sheduling policy/discipline)

我们对操作系统中运行的进程 (统称为工作负载, workload) 做一些简化假设:

- 每一个工作运行相同的时间
- 所有的工作同时到达

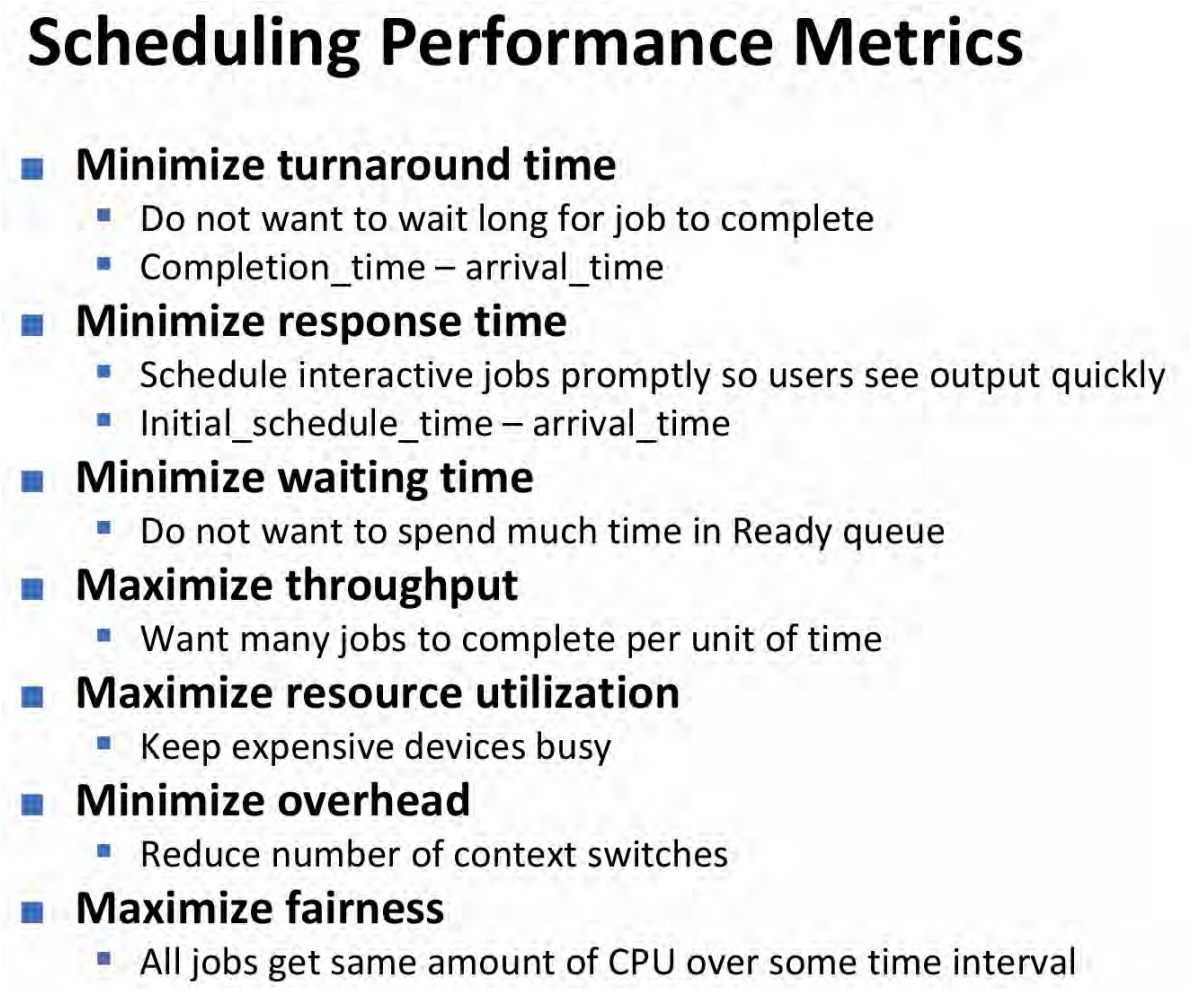
- 一旦开始，每个工作保持运行直到完成
- 所有的工作只用 CPU (即它们不执行 $\mathrm{I/O}$ 操作)
- 每个工作的运行时间是已知的

为简单起见，我们先只使用一个指标: **周转时间** (turnaround time)
其定义为任务完成时间减去任务到达系统的时间:

$$T_{\text{turn-around}} = T_{\text{complete}} - T_{\text{arrival}}$$

由于我们假设所有任务在同一时间到达，故 $T_{\text{arrival}} = 0$，于是有 $T_{\text{turn-around}} = T_{\text{complete}}$
随着我们放宽上述假设，这个情况将发生改变.

值得注意的是，周转时间是一个**性能指标** (performance metric)，这将是本章的首要关注点.
另一个有趣的指标是**公平** (fairness) (即在小规模的时间内将 CPU 均匀分配到工作负载之间)，比如 Jian's Fairness Index.
性能和公平在调度系统中往往是矛盾的.



(1) 先进先出 (FIFO)

最基本的调度算法即为**先进先出** (First In First Out, FIFO) 调度.
它很简单，易于实现.

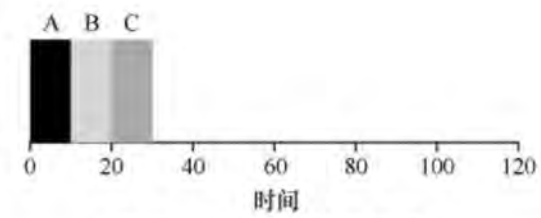


图 7.1　FIFO 的简单例子

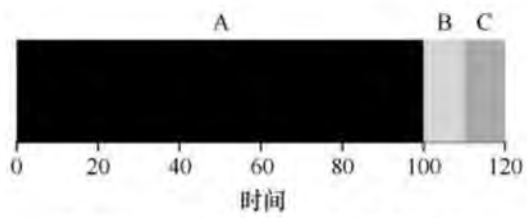


图 7.2　为什么 FIFO 没有那么好

但 FIFO 算法存在**护航效应** (convoy effect)
一些耗时较少的工作负载可能被排在耗时较长的工作负载之后，这会造成平均周转时间居高不下.
(Turnaround time can suffer when short jobs must wait for long jobs)

(2) 最短任务优先 (SJF)

解决 FIFO 护航效应的方法非常简单: **最短任务优先** (Shortest Job First, SJF)
在考虑平均周转时间的情况下，SJF 调度策略会比 FIFO 策略更好.
事实上，在所有工作负载同时到达的假设下，我们可以证明 SJF 是一个最优的调度算法.
但如果工作负载随时都可到达，而不是同时到达，则仍会出现问题:

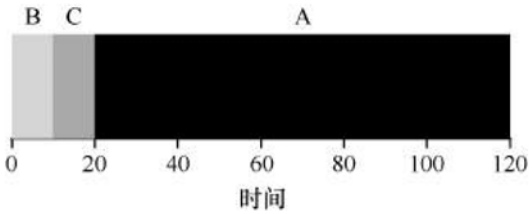


图 7.3　SJF 的简单例子

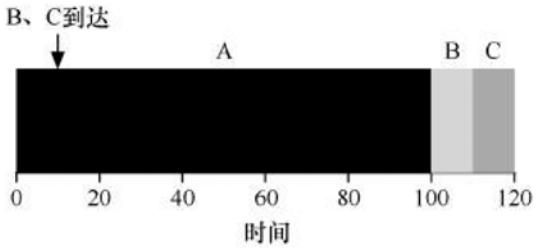


图 7.4　B 和 C 晚到时的 SJF

从图中可以看出，即使 B 和 C 在 A 之后不久到达，它们仍然被迫等到 A 完成，从而遭遇同样的护航问题.

(3) 最短完成时间优先 (STCF)

为解决 SJF 的问题，需要放宽假设条件 "每个工作保持运行直到完成"
回忆起我们先前关于时钟中断和上下文切换的讨论，
当 B 和 C 到达时，调度程序当然可以做其他事情:
它可以**抢占** (preempt) 工作 A，并决定运行另一个工作，或许稍后继续工作 A

> 根据我们的定义，SJF 是一种**非抢占式** (non-preemptive) 调度程序，因此存在上述问题.

Preemptive Scheduling

- **Prev schedulers:**
  - FIFO and SJF are non-preemptive
  - Only schedule new job when previous job voluntarily relinquishes CPU (performs I/O or exits)
- **New scheduler:**
  - **Preemptive**: Potentially schedule different job at any point by taking CPU away from running job
  - **STCF (Shortest Time-to-Completion First)**
  - Always run job that will complete the quickest

有一个调度程序就是这样做的: 向 SJF 添加抢占，称为**最短完成时间优先** (Shortest Time-to-Completion First, STCF)
每当新工作进入系统时，它就会确定剩余工作和新工作中，谁的剩余时间最少，然后调度该工作.

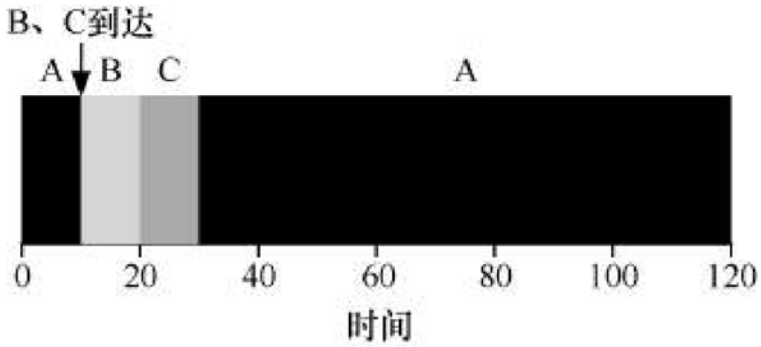
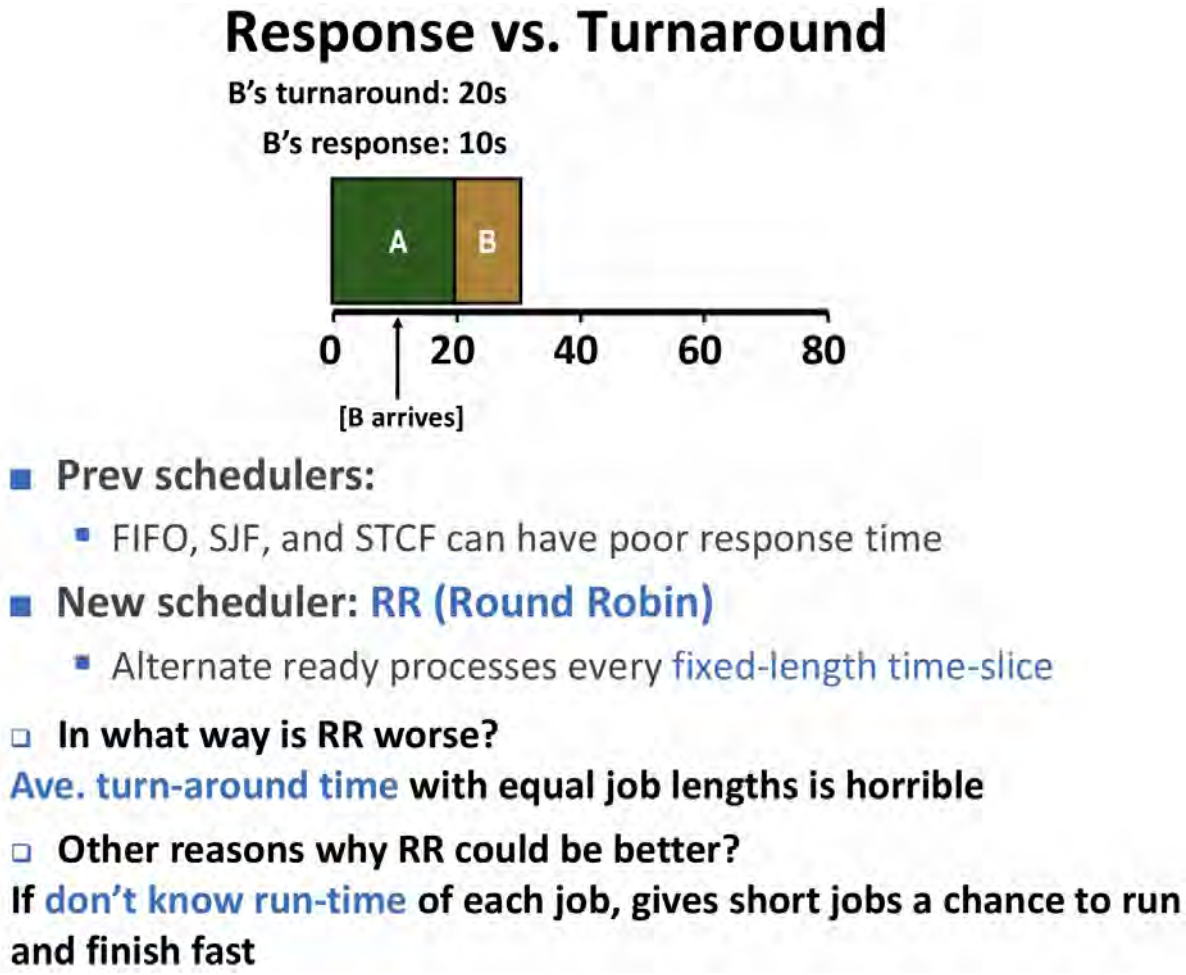


图 7.5　STCF 的简单例子

考虑到用户会坐在终端前面，因此我们需要要求系统的交互性好.
因此一个新的度量指标诞生了: **响应时间** (response time)
其定义为任务到达系统到首次运行的时间:

$$T_{\text{response}} = T_{\text{first-run}} - T_{\text{arrival}}$$

尽管 STCF 和相关方法有很好的平均周转时间，但其响应时间和交互性是相当糟糕的.



(4) 轮转 (RR)

为解决 STCF 和相关方法响应时间较慢的问题，我们介绍一种新的调度算法: **轮转** (Round-Robin, RR)
基本思想很简单:
RR 在一个**时间片** (time slice) 内运行一个工作，然后切换到运行队列中的下一个任务，而不是运行一个任务直到结束.
它反复执行，直到所有任务完成.
请注意，时间片长度必须是时钟中断周期的倍数.

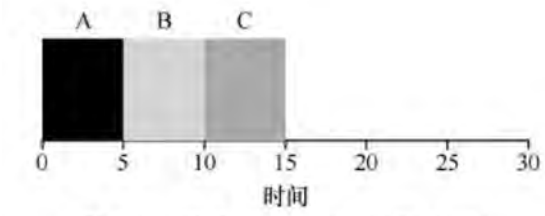


图 7.6　又是 SJF（响应时间不好）

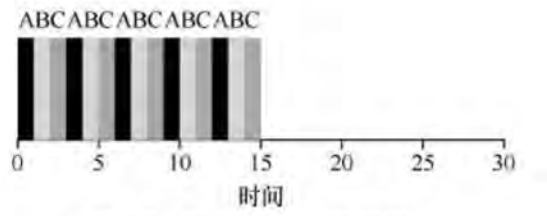


图 7.7　轮转（响应时间好）

时间片长度对于 RR 是至关重要的.
时间片长度越短，RR 在响应时间上表现越好.
然而时间片太短也是有问题的: 频繁进行上下文切换的成本将影响整体性能.
因此我们需要权衡时间片的长度，使其足够长以便**摊销** (amortize) 上下文切换成本，而又不会使系统不及时响应.

如果响应时间是我们的唯一指标，那么具有合理的时间片长度的 RR 就会是非常好的调度程序.
但是周转时间只关心作业何时完成，从这个角度来说 RR 几乎是最差的，在很多情况下甚至比简单的 FIFO 更差.
事实上，任何**公平** (fair) 的政策 (如 RR) 在周转时间这类指标上都表现不佳.

这是固有的权衡:
如果你愿意不公平，你可以运行较短的工作直到完成，但是要以响应时间为代价;
如果你重视公平性，则响应时间会较短，但会以周转时间为代价.

还有两个假设需要放宽:
假设 4 (作业没有 I/O 操作) 和假设 5 (每个作业的运行时间是已知的)

(5) 结合 I/O

调度程序显然要在工作发起 I/O 请求时做出决定，
因为当前正在运行的作业在 I/O 期间不会使用 CPU，它被阻塞等待 I/O 完成.
此时调度程序应该在 CPU 上安排另一项工作.
这样的**重叠** (overlap) 操作可以最大限度地提高系统的利用率.
当某些交互式作业正在执行 I/O 时，其他 CPU 密集型作业将运行，从而更好地利用处理器.

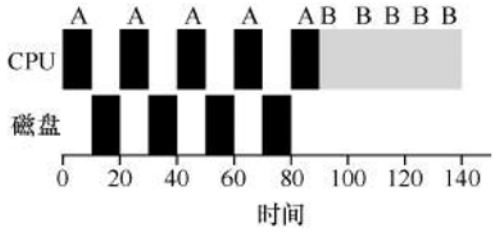


图 7.8　资源的糟糕使用

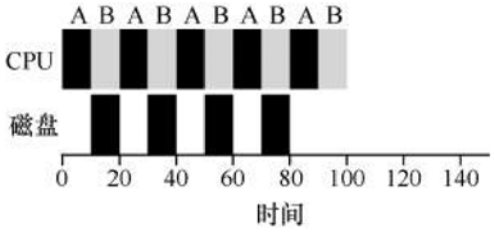


图 7.9　重叠可以更好地使用资源

调度程序还必须在 I/O 完成时做出决定.
发生这种情况时，会产生中断，操作系统运行并将发出 I/O 的进程从阻塞状态移回就绪状态.
当然它也可以决定在那个时候直接开始运行该项工作.

(5) 无法预知运行时间

我们放宽最后的假设 "调度程序知道每个工作的长度"
操作系统通常对每个作业的长度知之甚少.
那么我们该如何建立一个没有这种先验知识的 SJF 或 STCF?
更进一步，我们该如何结合 RR 调度程序，使得响应时间也变得相当不错呢?

1.4.2 多级反馈队列 (MLFQ)

本章将介绍一种著名的调度方法——**多级反馈队列** (Multi-level Feedback Queue, MLFQ)
多级反馈队列需要解决两方面的问题:

- 首先，它要优化周转时间，这通过先执行短工作来实现
- 其次，它要优化响应时间，这通过使用轮转策略来实现

但问题的关键在于:
通常我们对进程一无所知，那么我们应该如何在运行过程中学习进程的特征，从而做出更好的调度决策?

> 多级反馈队列是用历史经验预测未来的一个典型的例子.
> 如果工作有明显的阶段性行为，那么这种方式会很有效.
> 当然它也可能出错，使得系统做出比一无所知的时候更糟的决定.

(1) 基本规则

MLFQ 中有许多独立的**队列** (queue)，每个队列有不同的**优先级** (priority level)
任何时刻，一个工作只能存在于一个队列中.
MLFQ 总是优先执行较高优先级的工作 (即在较高级队列中的工作)
而对于具有同样的优先级的多个工作，MLFQ 采用轮转调度.

这就是 MLFQ 的两条基本规则:

- 规则 1: 如果 A 的优先级 $>$ B 的优先级，则运行 A (不运行 B)
- 规则 2: 如果 A 的优先级 $=$ B 的优先级，则轮转运行 A 和 B

因此 MLFQ 调度策略的关键在于如何设置优先级.

MLFQ 没有为每个工作预先指定不变的优先级，而是根据观察到的行为调整它的优先级.
例如，如果一个工作不断放弃 CPU 去等待键盘输入，这是交互型进程的可能行为，MLFQ 会让它保持高优先级
相反，如果一个工作长时间地占用 CPU，MLFQ 会降低其优先级.
通过这种方式，MLFQ 在进程运行过程中学习其行为，从而利用经验来预测它未来的行为.

但是我们容易预见到这样一个问题:

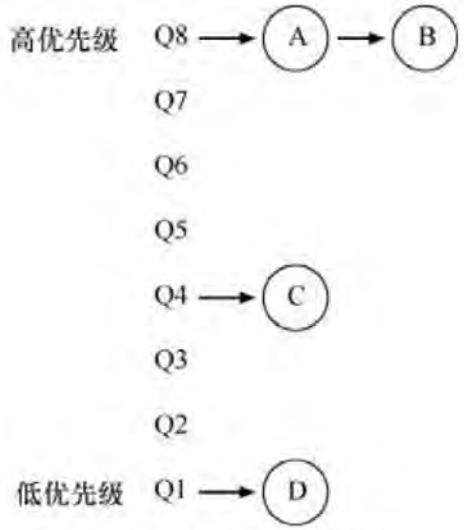


图 8.1　MLFQ 的例子

在上图给出的例子中，由于 A, B 具有最高优先级，调度程序会交替地调度它们.
可怜的 C 和 D 永远都没有机会运行，太气人了!
但幸运的是，上述例子只是一些队列的瞬时状态，工作的优先级是随时间动态调整的.

(2) 如何改变优先级

我们必须决定，在一个工作的生命周期中，MLFQ 如何改变其优先级 (在哪个队列中)

- 规则 3: 工作进入系统时，放在最高优先级 (最上层队列)
- 规则 $4a$: 工作用完整个时间片后，降低其优先级 (移入下一个队列)
- 规则 $4b$: 若工作在其时间片内主动释放 CPU，则其优先级不变.

示例一: 单个长工作

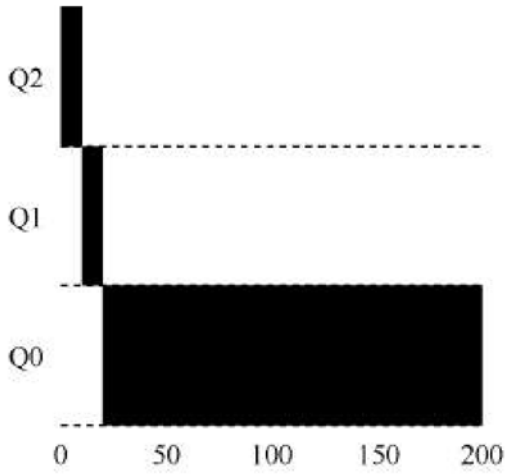


图 8.2　长时间工作随时间的变化

示例二: 单个长工作中途来了一个短工作
(这展示了 MLFQ 是如何近似 SJF 的)

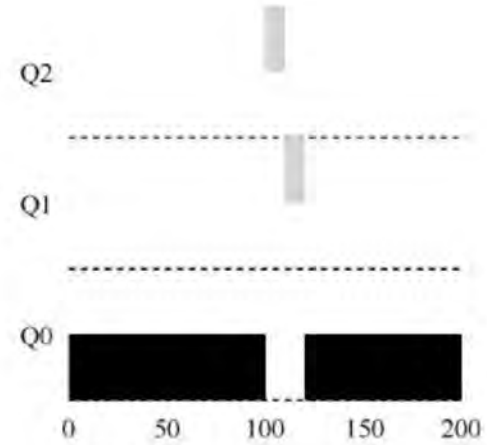


图 8.3　一个交互型工作

通过这个例子，你大概可以体会到这个算法的一个基本思想:
如果不知道工作是短工作还是长工作，那么就在开始的时候假设其是短工作，并赋予最高优先级.
如果确实是短工作，则很快会执行完毕，否则将被慢慢移入低优先级队列，而这时该工作也被认为是长工作了.
通过这种方式，MLFQ 近似于 SJF

示例三: 存在 $\mathrm{I/O}$ 的情况
回忆起规则 $4b$: 如果进程在时间片用完之前主动放弃 CPU，则保持它的优先级不变.
其意图很简单:
假设交互型工作中有大量的 $\mathrm{I/O}$ 操作 (比如等待用户的键盘或鼠标输入)，它会在时间片用完之前放弃 CPU.
在这种情况下，我们不想惩罚它，只是保持它的优先级不变.

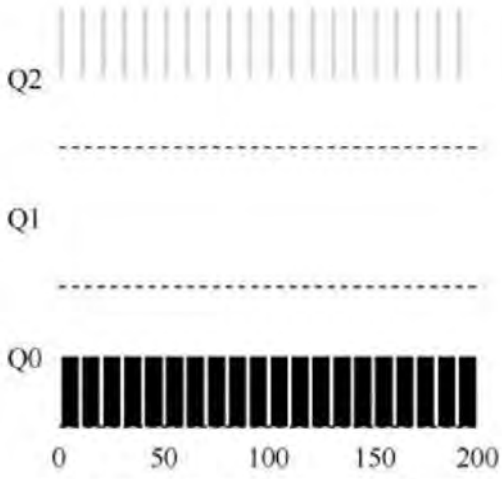


图 8.4　混合 I/O 密集型和 CPU 密集型工作负载

至此，我们有了基本的 MLFQ
它看起来似乎相当不错，长工作之间可以公平地分享 CPU，又能给短工作或交互型工作很好的响应时间.
然而它仍有一些非常严重的缺点:

- 首先会有**饥饿** (starvation) 问题.
  如果系统有太多交互型工作，就会不断占用 CPU，导致长工作永远无法得到 CPU (它们饿死了)
  即使在这种情况下，我们希望这些长工作也能有所进展
- 其次，聪明的用户会**愚弄**调度程序 (game the scheduler)
  即使用一些卑鄙的手段欺骗调度程序，让它给你远超公平的资源.
  上述算法对如下的攻击束手无策:
  进程在时间片用完之前，调用一个 I/O 操作 (比如访问一个无关的文件)，从而主动释放 CPU
  如此便可以保持在高优先级，占用更多的 CPU 时间.
- 最后，这个策略是**无情的** (unforgiving)
  一个程序可能在不同时间表现不同.
  一个计算密集的进程可能在某段时间表现为一个交互型的进程.
  如果使用我们目前的方法，它就不会享受系统中其他交互型工作的待遇.

(3) 提升优先级

要让 CPU 密集型工作也能取得一 些进展 (即使不多)，我们能做些什么?
一个简单的思路是周期性地**提升** (boost) 所有工作的优先级.
简单起见，我们将所有工作扔到最高优先级队列，于是有了如下的新规则:

- 规则 5: 每经过一段时间 T_{boost}，就将系统中所有工作重新加入最高优先级队列.

新规则一下解决了两个问题:

- 首先，进程不会饿死——在最高优先级队列中，它会以轮转的方式，与其他高优先级工作分享 CPU，从而最终获得执行
- 其次，如果一个 CPU 密集型工作变成了交互型，那么当它优先级提升时，调度程序会正确对待它

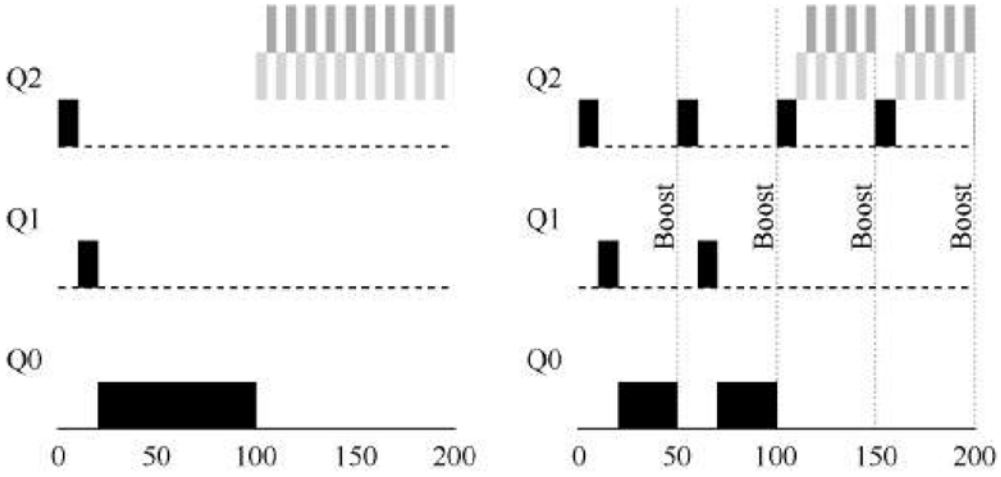


图 8.5　不采用优先级提升（左）和采用（右）

(4) 更好的计时方式

现在还有一个问题要解决: 如何阻止调度程序被愚弄?
可以看出，这个问题的元凶是规则 $4a$ 和 $4b$:

- 规则 $4a$: 工作用完整个时间片后，降低其优先级 (移入下一个队列)
- 规则 $4b$: 若工作在其时间片内主动释放 CPU，则其优先级不变.

这里的解决方案，是为 MLFQ 的每层队列提供更完善的 CPU 计时方式 (accounting)
调度程序应该记录一个进程在某一层队列中消耗的总时间，而不是在调度时重新计时.
只要进程用完了自己的配额，就将它降到低一优先级的队列中去，不论它是一次用完的，还是拆成很多次用完的.
为此，我们重写规则 $4a$ 和 $4b$:

- 规则 4: 一旦工作用完了其在某一层队列中的时间配额 (无论中间主动放弃了多少次 CPU)，就降低其优先级 (移入低一级队列)

在没有规则 4 的保护时，进程可以在每个时间片结束前发起一次 $\mathrm{I/O}$ 操作，从而垄断 CPU 时间.
有了这样的保护后，不论进程的 $\mathrm{I/O}$ 行为如何，都会慢慢地降低优先级，因而无法获得超过公平的 CPU 时间比例.

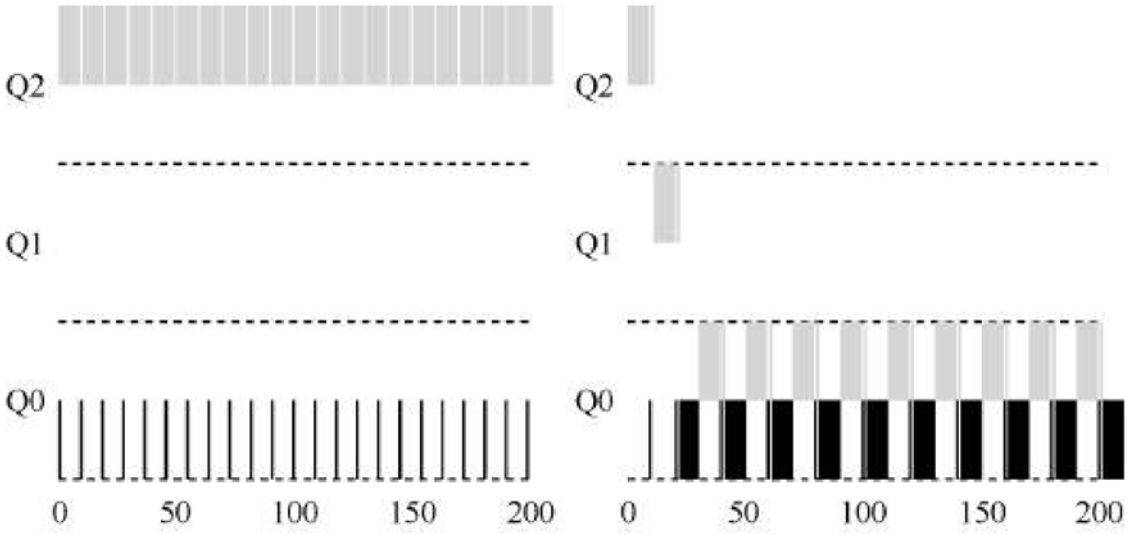


图 8.6　不采用愚弄反制（左）和采用（右）

(5) 小结

多级反馈队列 (MLFQ)——它有多级队列，并利用反馈信息决定某个工作的优先级.
以史为鉴: 关注进程的一贯表现，然后区别对待.

最终的 MLFQ 规则总结如下:

- 规则 1: 如果 A 的优先级 $>$ B 的优先级，则运行 A (不运行 B)
- 规则 2: 如果 A 的优先级 $=$ B 的优先级，则轮转运行 A 和 B
- 规则 3: 工作进入系统时，放在最高优先级 (最上层队列)
- 规则 4: 一旦工作用完了其在某一层队列中的时间配额 (无论中间主动放弃了多少次 CPU)，就降低其优先级 (移入低一级队列)
- 规则 5: 每经过一段时间 T_{boost}，就将系统中所有工作重新加入最高优先级队列.

MLFQ 不需要对工作的运行方式有先验知识，而是通过观察工作的运行来给出对应的优先级.
通过这种方式，MLFQ 可以同时满足各种工作的需求:
对于短时间运行的 $\mathrm{I/O}$ 密集型负载可以表现出接近于 $\mathrm{SJF/STCF}$ 的很好的全局性能，
同时对长时间运行的 CPU 密集型负载也可以公平地、不断地稳步向前.

1.4.3 彩票调度

我们来看一个不同类型的调度程序——比例份额 (proportional-share) 调度程序.
它的目标很简单: 确保每个工作获得一定比例的 CPU 时间，而不是优化周转时间和响应时间.

比例份额调度程序有一个非常优秀的现代例子，称为**彩票调度** (lottery scheduling)
基本思想很简单:
每隔一段时间，都会举行一次彩票抽奖，以确定接下来应该运行哪个进程.
越是应该频繁运行的进程，越是应该拥有更多地赢得彩票的机会.

(1) 彩票数代表份额

彩票数 (ticket) 代表了进程占有某个资源的份额.
一个进程拥有的彩票数占总彩票数的百分比，就是它占有资源的份额.
通过不断定时地 (例如每个时间片) 抽取彩票从概率上获得这种份额比例.

彩票调度最精彩的地方在于利用了**随机性** (randomness)
随机方法相对于传统的决策方式有以下优势:

- 随机方法通常不会出现极端情况 (最差情况)
- 随机方法很轻量, 几乎不需要记录任何状态 (只需记录每个进程拥有的彩票号码)
- 只要能很快地产生随机数，做出决策就很快 (当然，越是需要快的计算速度，随机就会越倾向于伪随机)
- 彩票调度的实现非常简单，只需要一个不错的随机数生成器来生成中奖号码
  以及一个记录系统中所有进程和彩票数的数据结构 (一个列表)
  我们只需从前往后遍历进程列表，用计数器计数，直至值超过中奖号码，当前列表元素对应的进程便是中奖者.
  要让这个过程更有效率，建议将列表项按照进程彩票数递减排序.
  这可保证用最小迭代次数找到需要的节点，尤其当大多数号码被少数进程掌握时.

```
// counter: used to track the cumulative sum of tickets
int counter = 0;
```

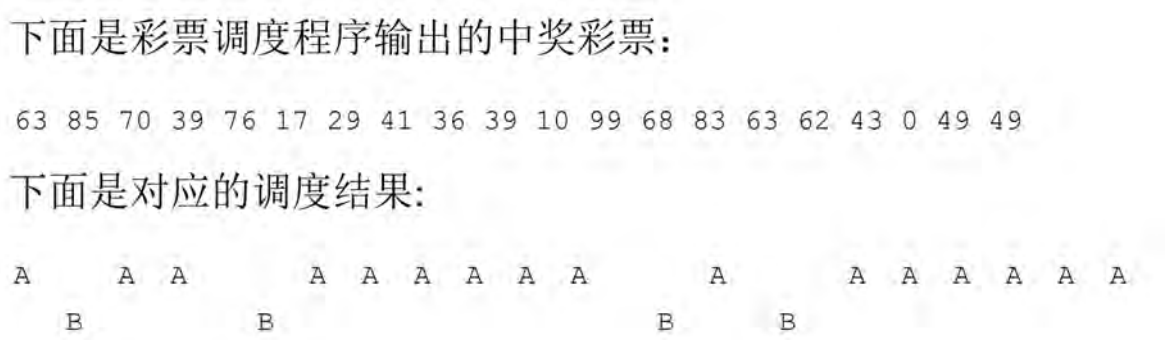
```c
// winner: generate a random value between 0 and the total number of tickets
int winner = getrandom(0, totaltickets - 1);  // totaltickets should be
greater than 0

// current: pointer to walk through the list of jobs (linked list)
node_t *current = head;

// loop through the job list until the sum of ticket values exceeds the
winner
while (current != NULL) // iterate through the list
{
    counter += current->tickets;  // accumulate tickets
    if (counter > winner)  // if the cumulative tickets exceed the winner
        break;  // found the winner
    current = current->next;  // move to the next node
}

// 'current' should point to the winning job, now schedule it
if (current != NULL)
    schedule(current);  // schedule the winning job
else
    // handle the case where no job is found (just in case)
    handle_error("No job found for the given ticket range.");
```

假设调度程序一共有 100 张彩票，进程 A 拥有 $0 \sim 74$ 这 75 个号码，进程 B 拥有 $75 \sim 99$ 这 25 个号码.
调度程序抽取彩票，对应 $0 \sim 99$ 中的一个数，拥有这个号码的进程中奖，在当前的时间片运行.



这两个工作运行的时间越长，它们得到的 CPU 时间比例就越接近期望.
我们可以调整进程所拥有的号码数使得不同工作接近同时完成.
但对于给定的一组工作，彩票分配问题依然没有最佳答案.

(2) 彩票机制

彩票调度还提供了一些机制，以不同且有效的方式来调度彩票:

- **① 彩票货币 (ticket currency)**
  这种方式允许拥有一组彩票的用户以他们喜欢的某种货币，将彩票分给自己的不同工作.
  之后操作系统再自动将这种货币兑换为全局彩票.

```
User A -> 500 (A's currency) to A1 -> 50 (global currency)
       -> 500 (A's currency) to A2 -> 50 (global currency)
User B -> 10 (B's currency) to B1 -> 100 (global currency)
```

- **② 彩票转让 (ticket transfer)**
  通过转让，一个进程可以临时将自己的彩票交给另一个进程.
  这种机制在客户端/服务端交互的场景中尤其有用.
  在这种场景中， 客户端进程向服务端发送消息，请求其按自己的需求执行工作.
  为了加速服务端的执行， 客户端可以将自己的彩票转让给服务端，从而尽可能加速服务端执行自己请求的速度.
  服务端执行结束后会将这部分彩票归还给客户端.
- **③ 彩票通胀 (ticket inflation)**
  利用通胀，一个进程可以临时提升或降低自己拥有的彩票数量.
  通胀可以用于进程之间相互信任的环境.
  在这种情况下，如果一个进程知道它需要更多 CPU 时间，就可以将自己的需求告知操作系统以增加自己的彩票.
  这一切不需要与任何其他进程通信.
  (当然在竞争环境中，进程之间互相不信任，这种机制就没什么意义.
  一个贪婪的进程可能给自己非常多的彩票，从而接管机器)

The End