

PSE51-docs

开发文档

PSE51 Documents

Version latest

中文

2025年12月08日

kurisaw.eu.org

2025, KurisaW

目录

- 1. overview
- 1.1. abort — 生成异常进程中止
- 1.2. abs
- 1.3. alarm
- 1.4. asctime — 将日期时间转换为字符串
- 1.5. asctime_r — 将日期和时间转换为字符串（已移除）
- 1.6. atof
- 1.7. atoi — 将字符串转换为整数
- 1.8. atol, atoll — 将字符串转换为长整型
- 1.9. atoll
- 1.10. bsearch — 二分搜索已排序的表
- 1.11. calloc — 内存分配器
- 1.12. clearerr
- 1.13. clock_getres, clock_gettime, clock_settime — 时钟和定时器函数
- 1.14. clock_getres, clock_gettime, clock_settime
- 1.15. clock_nanosleep
- 1.16. clock_settime
- 1.17. close, posix_close
- 1.18. confstr — 获取可配置字符串变量
- 1.19. ctime — 将时间值转换为日期和时间字符串
- 1.20. ctime_r — 将时间值转换为日期和时间字符串（已移除）
- 1.21. difftime — 计算两个日历时间值之间的差值
- 1.22. div
- 1.23. exit
- 1.24. fclose
- 1.25. fdatasync
- 1.26. fdopen

- 1.27. `feclearexcept` — 清除浮点异常
- 1.28. `fegetenv`, `fesetenv` — 获取和设置当前浮点环境
- 1.29. `fegetexceptflag`, `fesetexceptflag` — 获取和设置浮点状态标志
- 1.30. `fegetround`, `fesetround` — 获取和设置当前舍入方向
- 1.31. `feholdexcept`
- 1.32. `feof`
- 1.33. `feraiseexcept` — 引发浮点异常
- 1.34. `ferror` — 测试流的错误指示器
- 1.35. `fegetenv`, `fesetenv` — 获取和设置当前浮点环境
- 1.36. `fegetexceptflag`, `fesetexceptflag` — 获取和设置浮点状态标志
- 1.37. `fegetround`, `fesetround` — 获取和设置当前舍入方向
- 1.38. `fetestexcept` — 测试浮点异常标志
- 1.39. `feupdateenv` — 更新浮点环境
- 1.40. `fflush` — 刷新流
- 1.41. `fgetc` — 从流中获取一个字节
- 1.42. `fgets`
- 1.43. `fileno` — 将流指针映射为文件描述符
- 1.44. `flockfile`, `ftrylockfile`, `funlockfile` — `stdio` 锁定函数
- 1.45. `fopen` — 打开流
- 1.46. `fprintf`
- 1.47. `fputc` — 向流写入一个字节
- 1.48. `fputs`
- 1.49. `fread`
- 1.50. `free` — 释放已分配的内存
- 1.51. `freopen`
- 1.52. `fscanf`
- 1.53. `fsync`
- 1.54. `ftrylockfile`
- 1.55. `flockfile`, `ftrylockfile`, `funlockfile` — `stdio` 锁定函数
- 1.56. `fwrite` — 二进制输出
- 1.57. `getc` — 从流中获取一个字节
- 1.58. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked`
- 1.59. `getchar`

- 1.60. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — 读写字符
- 1.61. `getenv`, `secure_getenv` — 获取环境变量的值
- 1.62. `gets`
- 1.63. `gmtime`, `gmtime_r` — 将时间值转换为分解的UTC时间
- 1.64. `gmtime`, `gmtime_r` — 将时间值转换为分解的UTC时间
- 1.65. `imaxabs` — 返回绝对值
- 1.66. `imaxdiv`
- 1.67. `isalnum`, `isalnum_l` — 测试字母数字字符
- 1.68. `isalpha`, `isalpha_l` — 测试字母字符
- 1.69. `isblank`, `isblank_l` — 测试空白字符
- 1.70. `iscntrl`, `iscntrl_l` - 测试控制字符
- 1.71. `isdigit`, `isdigit_l` — 测试十进制数字字符
- 1.72. `isgraph`, `isgraph_l` — 测试可见字符
- 1.73. `islower`, `islower_l` - 测试小写字母
- 1.74. `isprint`, `isprint_l` — 测试可打印字符
- 1.75. `ispunct`, `ispunct_l` — 测试标点符号字符
- 1.76. `isspace`, `isspace_l` — 测试空白字符
- 1.77. `isupper`, `isupper_l` — 测试大写字母
- 1.78. `isxdigit`, `isxdigit_l` — 测试十六进制数字字符
- 1.79. `kill`
- 1.80. `labs`, `llabs` — 返回长整型绝对值
- 1.81. `ldiv`, `lldiv` — 计算长整型除法的商和余数
- 1.82. `llabs`
- 1.83. `lldiv`
- 1.84. `localeconv` — 返回本地化特定信息
- 1.85. `localtime`, `localtime_r` — 将时间值转换为本地时间的分解表示
- 1.86. `localtime`, `localtime_r` - 将时间值转换为分解的本地时间
- 1.87. `longjmp` — 非本地跳转
- 1.88. `malloc` — 内存分配器
- 1.89. `memchr`
- 1.90. `memcmp` — 内存字节比较
- 1.91. `memcpy` — 内存字节复制
- 1.92. `memmove`

- 1.93. memset
- 1.94. mktime — 将分解时间转换为自纪元以来的时间
- 1.95. mlock, munlock — 锁定或解锁进程地址空间范围 (REALTIME)
- 1.96. mlockall
- 1.97. mmap
- 1.98. munlock
- 1.99. munmap — 取消内存页映射
- 1.100. nanosleep — 高精度睡眠
- 1.101. open, openat — 打开文件
- 1.102. pause
- 1.103. perror
- 1.104. printf
- 1.105. pthread_atfork
- 1.106. pthread_attr_destroy
- 1.107. pthread_attr_getdetachstate, pthread_attr_setdetachstate — 获…
- 1.108. pthread_attr_getguardsize, pthread_attr_setguardsize — 获取和…
- 1.109. pthread_attr_getinheritsched, pthread_attr_setinheritsched — …
- 1.110. pthread_attr_getschedparam, pthread_attr_setschedparam
- 1.111. pthread_attr_getschedpolicy, pthread_attr_setschedpolicy
- 1.112. pthread_attr_getscope, pthread_attr_setscope
- 1.113. pthread_attr_getstack, pthread_attr_setstack
- 1.114. pthread_attr_getstackaddr, pthread_attr_setstackaddr
- 1.115. pthread_attr_getstacksize, pthread_attr_setstacksize — 获取和设…
- 1.116. pthread_attr_init, pthread_attr_destroy - 初始化和销毁线程属性对象
- 1.117. pthread_attr_getdetachstate, pthread_attr_setdetachstate — 获…
- 1.118. pthread_attr_getguardsize
- 1.119. pthread_attr_getinheritsched, pthread_attr_setinheritsched — …
- 1.120. pthread_attr_getschedparam, pthread_attr_setschedparam — 获…
- 1.121. pthread_attr_getschedpolicy
- 1.122. pthread_attr_getscope, pthread_attr_setscope — 获取和设置争用…
- 1.123. pthread_attr_getstack, pthread_attr_setstack — 获取和设置栈属性
- 1.124. pthread_attr_getstackaddr, pthread_attr_setstackaddr
- 1.125. pthread_attr_getstacksize, pthread_attr_setstacksize — 获取和设…

- 1.126. `pthread_cancel`
- 1.127. `pthread_cleanup_pop`, `pthread_cleanup_push`
- 1.128. `pthread_cleanup_push`, `pthread_cleanup_pop`
- 1.129. `pthread_cond_broadcast`, `pthread_cond_signal` — 广播或信号通…
- 1.130. `pthread_cond_destroy`, `pthread_cond_init`
- 1.131. `pthread_cond_destroy`, `pthread_cond_init` — 销毁和初始化条件变量
- 1.132. `pthread_cond_broadcast`, `pthread_cond_signal` — 广播或信号通…
- 1.133. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_co…`
- 1.134. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_co…`
- 1.135. `pthread_condattr_destroy`, `pthread_condattr_init`
- 1.136. `pthread_condattr_getclock`, `pthread_condattr_setclock`
- 1.137. `pthread_condattr_destroy`, `pthread_condattr_init`
- 1.138. `pthread_condattr_getclock`, `pthread_condattr_setclock`
- 1.139. `pthread_create` - 线程创建
- 1.140. `pthread_detach` — 分离线程
- 1.141. `pthread_equal` — 比较线程 ID
- 1.142. `pthread_exit` — 线程终止
- 1.143. `pthread_getconcurrency`
- 1.144. `pthread_getcpu`
- 1.145. `pthread_getschedparam`, `pthread_setschedparam` — 动态线程调…
- 1.146. `pthread_getspecific`, `pthread_setspecific` — 线程特定数据管理
- 1.147. `pthread_join` — 等待线程终止
- 1.148. `pthread_key_create` — 线程特定数据键创建
- 1.149. `pthread_key_delete`
- 1.150. `pthread_kill`
- 1.151. `pthread_mutex_destroy`
- 1.152. `pthread_mutex_getprioceiling`, `pthread_mutex_setprioceiling`
- 1.153. `pthread_mutex_destroy`, `pthread_mutex_init`
- 1.154. `pthread_mutex_lock`
- 1.155. `pthread_mutex_getprioceiling`, `pthread_mutex_setprioceiling`
- 1.156. `pthread_mutex_trylock` - 尝试锁定互斥锁
- 1.157. `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_u…`
- 1.158. `pthread_mutexattr_destroy`

- 1.159. `pthread_mutexattr_getprioceiling`, `pthread_mutexattr_setpriocei...`
- 1.160. `pthread_mutexattr_getprotocol`
- 1.161. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` — 获取...
- 1.162. `pthread_mutexattr_init`, `pthread_mutexattr_destroy` - 销毁和初始...
- 1.163. `pthread_mutexattr_getprioceiling`, `pthread_mutexattr_setpriocei...`
- 1.164. `pthread_mutexattr_setprotocol`, `pthread_mutexattr_getprotocol`
- 1.165. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` - 获取和...
- 1.166. `pthread_once` — 动态包初始化
- 1.167. `pthread_self` — 获取调用线程ID
- 1.168. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcan...`
- 1.169. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcan...`
- 1.170. `pthread_setconcurrency`, `pthread_getconcurrency`
- 1.171. `pthread_getschedparam`, `pthread_setschedparam`
- 1.172. `pthread_setschedprio`
- 1.173. `pthread_getspecific`, `pthread_setspecific` — 线程特定数据管理
- 1.174. `pthread_sigmask`, `sigprocmask` — 检查和更改阻塞信号
- 1.175. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcan...`
- 1.176. `putc` — 向流中写入一个字节
- 1.177. `putc_unlocked`
- 1.178. `putchar`
- 1.179. `putchar_unlocked`
- 1.180. `puts`
- 1.181. `qsort`, `qsort_r` — 数据表排序
- 1.182. `raise`
- 1.183. `rand`, `srand` — 伪随机数生成器
- 1.184. `rand`, `rand_r`, `srand` - 伪随机数生成器
- 1.185. `read`
- 1.186. `realloc`, `reallocarray` — 内存重新分配器
- 1.187. `scanf`
- 1.188. `sched_get_priority_max`, `sched_get_priority_min`
- 1.189. `sched_get_priority_max`, `sched_get_priority_min`
- 1.190. `sched_rr_get_interval` — 获取执行时间限制 (REALTIME)
- 1.191. `sem_close` — 关闭命名信号量

- 1.192. `sem_destroy` — 销毁无名信号量
- 1.193. `sem_getvalue`
- 1.194. `sem_init` — 初始化匿名信号量
- 1.195. `sem_open` — 初始化并打开命名信号量
- 1.196. `sem_post` — 解锁信号量
- 1.197. `sem_clockwait`, `sem_timedwait` — 锁定信号量
- 1.198. `sem_trywait`, `sem_wait` — 锁定信号量
- 1.199. `sem_unlink` - 删除命名信号量
- 1.200. `sem_trywait`, `sem_wait` — 锁定信号量
- 1.201. `setbuf`
- 1.202. `setenv` — 添加或更改环境变量
- 1.203. `setjmp`
- 1.204. `setlocale` — 设置程序本地化环境
- 1.205. `setvbuf` — 为流分配缓冲区
- 1.206. `shm_open` — 打开共享内存对象 (REALTIME)
- 1.207. `shm_unlink` — 删除共享内存对象 (REALTIME)
- 1.208. `sigaction`
- 1.209. `sigaddset`
- 1.210. `sigdelset` - 从信号集中删除一个信号
- 1.211. `sigemptyset`
- 1.212. `sigfillset` - 初始化并填充信号集
- 1.213. `sigismember`
- 1.214. `signal` — 信号管理
- 1.215. `sigpending`
- 1.216. `sigprocmask`
- 1.217. `sigqueue`
- 1.218. `sigsuspend` — 等待信号
- 1.219. `sigtimedwait`, `sigwaitinfo`
- 1.220. `sigwait` — 等待排队的信号
- 1.221. `sigtimedwait`, `sigwaitinfo` — 等待排队信号
- 1.222. `snprintf`, `asprintf`, `dprintf`, `fprintf`, `printf`, `sprintf` — 打印格式化输出
- 1.223. `sprintf` - 打印格式化输出
- 1.224. `rand`, `srand` — 伪随机数生成器

- 1.225. `fscanf`, `scanf`, `sscanf`
- 1.226. `strcat`
- 1.227. `strchr`
- 1.228. `strcmp` — 比较两个字符串
- 1.229. `strcoll`
- 1.230. `strcpy`
- 1.231. `strcspn`
- 1.232. `strerror`, `strerror_l`, `strerror_r` — 获取错误消息字符串
- 1.233. `strerror`, `strerror_l`, `strerror_r` — 获取错误消息字符串
- 1.233.1. 1 和 #2 的组合: `errno` 被设置为 `[EINVAL]`, 返回值指向类似 "un..."
- 1.234. `strftime`, `strftime_l` — 将日期和时间转换为字符串
- 1.235. `strlen`, `strnlen` — 获取固定大小字符串的长度
- 1.236. `strncat` — 将一个字符串的部分内容连接到另一个字符串末尾
- 1.237. `strcmp`
- 1.238. `strncpy`
- 1.239. `strpbrk`
- 1.240. `strrchr` — 字符串扫描操作
- 1.241. `strspn` — 获取子字符串长度
- 1.242. `strstr` — 查找子字符串
- 1.243. `strtod`, `strtof`, `strtold` — 将字符串转换为双精度浮点数
- 1.244. `strtod`, `strtof`, `strtold` - 将字符串转换为双精度浮点数
- 1.245. `strtoimax`, `strtoumax` — 将字符串转换为整数类型
- 1.246. `strtok`, `strtok_r` — 将字符串分割为标记
- 1.247. `strtok`, `strtok_r` — 将字符串分割为标记
- 1.248. `strtol`, `strtoll` — 将字符串转换为长整型
- 1.249. `strtold`
- 1.250. `strtoll`
- 1.251. `strtoul`
- 1.252. `strtoull` — 将字符串转换为无符号长长整型
- 1.253. `strtoumax`
- 1.254. `strxfrm`, `strxfrm_l` — 字符串转换
- 1.255. `sysconf`
- 1.256. `time` — 获取时间

- 1.257. timer_create - 创建每进程定时器
- 1.258. timer_delete
- 1.259. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器
- 1.260. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器
- 1.261. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器
- 1.262. tolower, tolower_l — 将大写字符转换为小写字符
- 1.263. toupper, toupper_l — 将小写字符转换为大写字符
- 1.264. tzset - 设置时区转换信息
- 1.265. uname
- 1.266. ungetc — 将字节推回输入流
- 1.267. unsetenv
- 1.268. va_arg
- 1.269. va_copy
- 1.270. va_end
- 1.271. va_start, va_arg, va_copy, va_end - 处理可变参数列表
- 1.272. vfprintf
- 1.273. vfscanf, vscanf, vsscanf — stdarg 参数列表的格式化输入
- 1.274. vprintf
- 1.275. vfscanf, vscanf, vsscanf — 格式化输入 stdarg 参数列表
- 1.276. vsprintf
- 1.277. vsprintf - 格式化 stdarg 参数列表的输出
- 1.277.1. 示例
- 1.277.2. 应用程序用法
- 1.277.3. 原理
- 1.277.4. 未来方向
- 1.277.5. 参见
- 1.277.6. 变更历史
- 1.278. vsscanf
- 1.279. write

1. overview

1.1. `abort` — 生成异常进程中止

概要

```
#include <stdlib.h>

_Noreturn void abort(void);
```

描述

`abort()` 函数应当导致异常进程终止发生，除非它生成的 SIGABRT 信号被捕获且信号处理器不返回。

异常终止处理应当包括为 SIGABRT 定义的默认操作，并且可能包括尝试对所有打开的流执行 `fclose()` 的操作。

SIGABRT 信号应当发送到调用线程，就如同通过使用参数 SIGABRT 调用 `raise()` 一样。如果此信号不终止进程（例如，如果信号被捕获且处理器返回），`abort()` 可以将 SIGABRT 的处理方式更改为 SIG_DFL 并再次发送信号（以相同方式）。如果发送了第二个信号且它不终止进程，则行为未指定，但 `abort()` 调用不应返回。

由 `abort()` 提供给 `wait()`、`waitid()` 或 `waitpid()` 的状态应当是进程被 SIGABRT 信号终止的状态。`abort()` 函数应当覆盖阻塞或忽略 SIGABRT 信号。

返回值

`abort()` 函数不应返回。

错误

未定义错误。

示例

无。

应用程序用法

捕获信号旨在为应用程序开发人员提供中止处理的可移植方式，不受任何实现提供函数的可能干扰。

基本原理

历史上，`abort()` 通过调用其他信号操作函数来实现，如 `raise()`、`sigaction()` 和 `pthread_sigmask()`。这意味着它的操作可能受到其他线程中并发操作的影响。例如，如果 `abort()` 尝试通过调用 `sigaction()` 将 SIGABRT 的处理方式更改为 SIG_DFL 然后调用 `raise()` 来终止进程，另一个线程可能在这两个调用之间更改处理方式，导致进程不被终止。如果发生这种情况，唯一的要求是 `abort()` 不返回。实现可以在循环中调用这些函数（理论上可能无限执行），或者可以通过调用 `_exit()` 来终止进程（这会确保终止但会导致错误的等待状态）。为了避免这些问题，鼓励实现以不受其他线程并发操作影响的方式实现 `abort()`。例如，它可以首先停止所有其他线程的执行，或者可以通过"如同通过信号终止"系统调用来终止进程，而不是通过引发（第二个）SIGABRT。

ISO/IEC 9899:1999 标准要求（当前标准仍然要求）`abort()` 函数是异步信号安全的。由于 POSIX.1-2024 遵循 ISO C 标准，这需要将描述中的"应包括 `fclose()` 的效果"更改为"可能包括尝试执行 `fclose()` 的操作"。

修订后的措辞允许一些向后兼容性，并避免潜在的死锁情况。

应用 Open Group 基础决议 bwg2002-003，从描述中删除以下 XSI 阴影段落：

"在 XSI 兼容系统上，此外异常终止处理应包括对消息目录描述符执行 `fclose()` 的效果。"

删除此段落有几个原因：

- 在异常进程终止之前不需要对打开的消息目录执行特殊处理。

- 特别提及 `abort()` 包括对打开流执行 `fclose()` 效果的主要原因是刷新流上排队的输出。此上下文中的消息目录是只读的，因此不需要刷新。
- 对消息目录描述符执行 `fclose()` 的效果是未指定的。允许但不要求使用文件描述符实现消息目录描述符，但在 POSIX.1-2024 中没有提及消息目录描述符使用标准 I/O 流 FILE 对象，而这正是 `fclose()` 所期望的。

未来方向

本标准的未来版本可能要求 `abort()` 以不受其他线程并发操作影响的方式实现。

另请参见

`_exit()`、`kill()`、`raise()`、`signal()`、`wait()`、`waitid()`
XBD `<stdlib.h>`

变更历史

首次发布于第 1 版。源自 SVID 的第 1 版。

第 6 版

标记了超出 ISO C 标准的扩展。

对描述进行了更改以与 ISO/IEC 9899:1999 标准对齐。

应用了 Open Group 基础决议 bwg2002-003。

应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/10，更改了异常终止处理的描述并在基本原理部分添加了内容。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/9，在应用程序用法部分将"实现定义的函数"更改为"实现提供的函数"。

第 8 版

应用了 Austin Group 缺陷 906，阐明了 `abort()` 的行为如何可能受到其他线程中并发操作的影响。

应用了 Austin Group 缺陷 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。

1.2. abs

SYNOPSIS

```
#include <stdlib.h>

int abs(int i);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

abs() 函数应计算其整数操作数 **i** 的绝对值。如果结果无法表示，则行为未定义。

RETURN VALUE

abs() 函数应返回其整数操作数的绝对值。

ERRORS

未定义任何错误。

以下小节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

由于 POSIX.1 要求 **int** 类型使用二进制补码表示，具有最大量级的负整数 **{INT_MIN}** 的绝对值无法表示，因此 **abs(INT_MIN)** 是未定义的。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [fabs\(\)](#)
- [labs\(\)](#)

XBD [<stdlib.h>](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 8

应用了 Austin Group Defect 1108，更改了 APPLICATION USAGE 小节。

1.3. alarm

SYNOPSIS

```
#include <unistd.h>

unsigned alarm(unsigned seconds);
```

DESCRIPTION

`alarm()` 函数应使系统在经过 `seconds` 参数指定的实时秒数后为进程生成 SIGALRM 信号。处理器调度延迟可能会阻止进程在信号生成时立即处理该信号。

如果 `seconds` 为 0，则取消任何挂起的闹钟请求。

闹钟请求不会堆叠；只能以这种方式调度一个 SIGALRM 信号生成。如果 SIGALRM 信号尚未生成，则调用将导致重新调度生成 SIGALRM 信号的时间。

RETURN VALUE

如果存在先前的 `alarm()` 请求且仍有剩余时间，`alarm()` 应返回一个非零值，该值是先前会生成 SIGALRM 信号的秒数。否则，`alarm()` 应返回 0。

ERRORS

`alarm()` 函数总是成功的，没有保留返回值来表示错误。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

`fork()` 函数会清除子进程中挂起的闹钟。由 `exec` 系列函数之一创建的新进程映像会继承旧进程映像中闹钟信号的剩余时间。

应用程序开发人员应注意，参数 `seconds` 和 `alarm()` 的返回值类型为 `unsigned`。这意味着严格符合的 POSIX 系统接口应用程序不能传递大于 `{UINT_MAX}` 最小保证值的值，ISO C 标准将其设置为 65535，任何传递更大值的应用程序都会限制其可移植性。曾考虑过使用不同的类型，但历史实现（包括具有 16 位 `int` 类型的实现）一致使用 `unsigned` 或 `int`。

应用程序开发人员应注意同一进程同时使用 `alarm()` 和 `sleep()` 函数时可能产生的交互。

RATIONALE

许多历史实现（包括 Version 7 和 System V）允许闹钟最多提前一秒触发。其他实现允许闹钟最多提前半秒或一个时钟节拍触发，或者根本不允许提前触发。后者被认为是最合适的，因为它提供了最可预测的行为，特别是因为信号由于调度原因总是可以被无限期延迟。因此，应用程序可以将 `seconds` 参数选择为它们希望在信号之前经过的最长时间量。

此处和其他地方（`sleep()`、`times()`）使用的术语“实时”意在表示日常英语用法中的“挂钟时间”，与“实时操作系统”无关。它与虚拟时间形成对比，如果仅使用时间一词，可能会被误解。

在某些实现中，包括 4.3 BSD，`seconds` 参数的非常大的值会被静默向下舍入为特定于实现的最大值。该最大值足够大（以几个月为单位），以至于效果不明显。

在多线程应用程序中，闹钟生成有两种可能的选择：为调用线程生成或为进程生成。第一个选项不会特别有用，因为闹钟状态是按进程维护的，并且由最后一次调用 `alarm()` 建立的闹钟是唯一活跃的。

此外，允许为线程生成异步信号会在整体信号模型中引入例外。这需要有令人信服的理由才能被证明是合理的。

FUTURE DIRECTIONS

无。

SEE ALSO

[alarm](#), [exec](#), [fork\(\)](#), [pause\(\)](#), [sigaction\(\)](#), [sleep\(\)](#), [timer_create\(\)](#)

XBD ,

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID Issue 1。

Issue 6

对 POSIX 实现的以下新要求源自与单一 UNIX 规范的对齐：

- 更新了 DESCRIPTION 以指明与 `setitimer()`、`ualarm()` 和 `usleep()` 函数的交互是未指定的。

应用了 IEEE Std 1003.1-2001/Cor 2-2004 项目 XSH/TC2/D6/16，在 RATIONALE 中将"实现定义的最大值"替换为"特定于实现的最大值"。

Issue 8

应用了 Austin Group 缺陷 1330，移除了过时的接口。

1.4. `asctime` — 将日期时间转换为字符串

概要

```
#include <time.h>

char *asctime(const struct tm *timeptr);
```

描述

`asctime()` 函数应将 `timeptr` 指向的结构体中的分解时间 (broken-down time) 转换为以下形式的字符串：

```
Sun Sep 16 01:03:52 1973\n\0
```

使用等效于以下算法的方式：

```
char *asctime(const struct tm *timeptr)
{
    static char wday_name[7][3] = {
        "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"
    };
    static char mon_name[12][3] = {
        "Jan", "Feb", "Mar", "Apr", "May", "Jun",
        "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"
    };
    static char result[26];

    sprintf(result, "%.3s %.3s%3d %.2d:%.2d:%.2d %d\n",
            wday_name[timeptr->tm_wday],
            mon_name[timeptr->tm_mon],
            timeptr->tm_mday, timeptr->tm_hour,
            timeptr->tm_min, timeptr->tm_sec,
            1900 + timeptr->tm_year);
    return result;
}
```

如果分解时间的任何成员包含超出其正常范围的值 (参见 XBD `<time.h>`)，`asctime()` 函数的行为是未定义的。同样，如果计算出的年份超过四位数字或小于年份 1000，行为也是未定义的。

`tm` 结构体在 `<time.h>` 头文件中定义。

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解时间结构体和一个 `char` 类型的数组。执行任何返回指向这些对象类型之一的指针的函数可能会覆盖从之前调用其中任何函数所返回的值指向的任何同类型对象中的信息。

`asctime()` 函数不需要是线程安全的；但是，`asctime()` 应避免与除自身、`ctime()`、`gmtime()` 和 `localtime()` 之外的所有函数发生数据竞争。

返回值

成功完成时，`asctime()` 应返回指向字符串的指针。如果函数不成功，应返回 `NULL`。

错误

未定义错误。

应用程序用法

此函数仅用于与较旧的实现保持兼容性。如果结果字符串过长，它具有未定义行为，因此应避免使用此函数。在未检测到输出字符串长度溢出的实现上，可能会以导致应用程序失败或可能违反系统安全的方式溢出输出缓冲区。此外，此函数不支持本地化的日期时间格式。为避免这些问题，应用程序应使用 `strftime()` 从分解时间生成字符串。

分解时间结构体的值可以通过调用 `gmtime()` 或 `localtime()` 获得。

基本原理

标准开发者决定将 `asctime()` 函数标记为过时的（obsolescent），尽管它在 ISO C 标准中，但存在缓冲区溢出的可能性。ISO C 标准也提供了 `strftime()` 函数，可用于避免这些问题。

未来方向

此函数可能在将来的版本中被移除，但要在从 ISO C 标准中移除之后才能移除。

参见

- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

更改历史

首次发布于 Issue 1

源自 SVID 的 Issue 1。

Issue 5

- 先前在 APPLICATION USAGE 部分中的规范性文本移至 DESCRIPTION。
- 为与 POSIX 线程扩展对齐，包含了 `asctime_r()` 函数。
- 在 DESCRIPTION 中添加了说明 `asctime()` 函数不需要是可重入的注释。

Issue 6

- `asctime_r()` 函数被标记为线程安全函数选项的一部分。
- 超出 ISO C 标准的扩展被标记。
- APPLICATION USAGE 部分更新为包含线程安全函数的注释及其避免可能使用静态数据区域的说明。

- `asctime_r()` 的 DESCRIPTION 更新为描述返回字符串的格式。
- 为与 ISO/IEC 9899:1999 标准对齐, 向 `asctime_r()` 原型添加了 `restrict` 关键字。
- 应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/17, 在 RETURN VALUE 部分添加了 CX 扩展, 要求如果 `asctime()` 函数不成功则返回 NULL。

Issue 7

- 应用 Austin Group Interpretation 1003.1-2001 #053, 将这些函数标记为过时的。
- 应用 Austin Group Interpretation 1003.1-2001 #156。
- `asctime_r()` 函数从线程安全函数选项移至 Base。
- 应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0033 [86,429]。

Issue 8

- 应用 Austin Group Defect 469, 阐明 `asctime()` 行为未定义的条件。
- 应用 Austin Group Defect 1302, 使此函数与 ISO/IEC 9899:2018 标准对齐。
- 应用 Austin Group Defect 1330, 更改 FUTURE DIRECTIONS 部分。
- 应用 Austin Group Defect 1376, 从源自 ISO C 标准的某些文本中移除 CX 着色并更新以匹配 ISO C 标准。
- 应用 Austin Group Defect 1410, 移除 `asctime_r()` 函数。

1.5. `asctime_r` — 将日期和时间转换为字符串（已移除）

状态

已移除: `asctime_r()` 函数已从 POSIX.1-2024 标准中移除 (Issue 8, Austin Group Defect 1410)。

历史信息

本页面记录了 `asctime_r()` 函数的历史背景, 该函数之前是 POSIX 标准的一部分, 但现已被移除。

名称 (历史)

`asctime_r` — 将日期和时间转换为字符串 (线程安全)

概要 (历史)

```
#include <time.h>

char *asctime_r(const struct tm *restrict timeptr, char *restrict
```

描述

此参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 版本遵循 ISO C 标准。

`asctime_r()` 函数是 `asctime()` 的线程安全版本, 旨在将 `timeptr` 指向的结构体中的分解时间转换为以下格式的字符串:

```
Sun Sep 16 01:03:52 1973\n\0
```

与 `asctime()` 的主要区别

- **线程安全性:** `asctime_r()` 是线程安全的, 而 `asctime()` 不是
- **缓冲区管理:** `asctime_r()` 要求调用者提供缓冲区
- **静态数据避免:** `asctime_r()` 避免使用静态数据区域

历史背景

在 POSIX 版本中的演进

Issue 5: `asctime_r()` 函数被纳入以与 POSIX 线程扩展保持一致。

Issue 6:

- `asctime_r()` 函数被标记为线程安全函数选项的一部分
- 为与 ISO/IEC 9899:1999 标准保持一致, 在 `asctime_r()` 原型中添加了 `restrict` 关键字
- 标记了超出 ISO C 标准的扩展
- 更新了应用程序使用部分, 包含关于线程安全函数的说明

Issue 7:

- `asctime_r()` 函数从线程安全函数选项移至基础标准
- 应用了 Austin Group Interpretation 1003.1-2001 #053, 将这些函数标记为过时

Issue 8:

- 应用了 Austin Group Defect 1410, 从标准中移除了 `asctime_r()` 函数

移除理由

标准开发者决定将 `asctime()` 函数 (并由此扩展到 `asctime_r()`) 标记为过时, 尽管它在 ISO C 标准中, 但由于可能存在缓冲区溢出的问题。ISO C 标准也提供了 `strftime()` 函数, 可以用来避免这些问题。

应用程序使用 (历史)

此函数仅为与旧实现兼容而包含。如果结果字符串过长，它的行为是未定义的，因此应不鼓励使用此函数。

推荐的替代方案：应用程序应使用 `strftime()` 从分解时间生成字符串，以避免缓冲区溢出问题并支持本地化的日期和时间格式。

分解时间结构体的值可以通过调用 `gmtime()` 或 `localtime()` 获得。

未来方向

`asctime_r()` 函数已被移除，在未来的 POSIX 标准版本中将不可用。

另见

- `asctime()` - 非线程安全版本（也已过时）
- `clock()`
- `ctime()`
- `difftime()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()` - **推荐的替代品**
- `strptime()`
- `time()`
- XBD `<time.h>`

变更历史

首次发布：Issue 1。源自 SVID 的 Issue 1。

Issue 5：

- 之前在应用程序使用部分中的规范性文本移至描述部分
- `asctime_r()` 函数被纳入以与 POSIX 线程扩展保持一致

- 在描述部分添加了说明 `asctime()` 函数不需要可重入的注释

Issue 6:

- `asctime_r()` 函数被标记为线程安全函数选项的一部分
- 标记了超出 ISO C 标准的扩展
- 应用程序使用部分更新为包含关于线程安全函数及其避免可能使用静态数据区域的说明
- 更新了 `asctime_r()` 的描述，描述返回字符串的格式
- 为与 ISO/IEC 9899:1999 标准保持一致，在 `asctime_r()` 原型中添加了 `restrict` 关键字
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/17，在返回值部分添加了 CX 扩展，要求如果 `asctime()` 函数不成功则返回 NULL

Issue 7:

- 应用了 Austin Group Interpretation 1003.1-2001 #053，将这些函数标记为过时
- 应用了 Austin Group Interpretation 1003.1-2001 #156
- `asctime_r()` 函数从线程安全函数选项移至基础标准
- 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0033 [86,429]

Issue 8:

- 应用了 Austin Group Defect 469，阐明了 `asctime()` 行为未定义的条件
 - 应用了 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准保持一致
 - 应用了 Austin Group Defect 1330，更改了未来方向部分
 - 应用了 Austin Group Defect 1376，从源自 ISO C 标准的某些文本中移除 CX 着色并更新以匹配 ISO C 标准
 - **应用了 Austin Group Defect 1410，移除了 `asctime_r()` 函数**
-

1.6. atof

概要

```
#include <stdlib.h>

double atof(const char *str);
```

描述

调用 `atof(str)` 应等效于：

```
strtod(str, (char **)NULL),
```

除了错误处理可能有所不同。如果值无法表示，则行为未定义。

返回值

`atof()` 函数返回转换后的值。

错误

未定义任何错误。

示例

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    const char *str = "123.456";
    double result = atof(str);
    printf("Converted value: %f\n", result);
    return 0;
}
```

应用程序使用

`atof()` 函数不要求是线程安全的。

基本原理

以下章节为参考性内容。

参考性文本结束。

1.7. atoi — 将字符串转换为整数

概要

```
#include <stdlib.h>

int atoi(const char *str);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非预期的。本卷 POSIX.1-2024 遵从 ISO C 标准。

调用 `atoi(str)` 应等效于：

```
(int) strtol(str, (char **)NULL, 10)
```

但错误处理可能有所不同。如果值无法表示，则行为未定义。

返回值

如果值可以表示，`atoi()` 函数应返回转换后的值。

错误

未定义任何错误。

以下章节为参考信息。

示例

转换参数

以下示例检查程序的使用是否正确。如果存在参数且此参数的十进制转换（使用 `atoi()` 获得）大于 0，则程序具有有效的等待事件的分钟数。

```
#include <stdlib.h>
#include <stdio.h>
...
int minutes_to_event;
...
if (argc < 2 || (minutes_to_event = atoi(argv[1])) <= 0) {
    fprintf(stderr, "Usage: %s minutes\n", argv[0]); exit(1);
}
...
```

应用程序使用

`atoi()` 函数已被 `strtol()` 包含，但由于在现有代码中广泛使用而保留。如果不知道数字是否在范围内，应使用 `strtol()`，因为 `atoi()` 不需要执行任何错误检查。

理由

无。

未来方向

无。

另请参阅

- `strtol()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 1541，更改了示例部分。

1.8. atol, atoll — 将字符串转换为长整型

概要

```
#include <stdlib.h>

long atol(const char *nptr);
long long atoll(const char *nptr);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

除下述说明外，`atol(nptr)` 调用应等价于：

```
strtol(nptr, (char **)NULL, 10)
```

除下述说明外，`atoll(nptr)` 调用应等价于：

```
strtoll(nptr, (char **)NULL, 10)
```

错误处理方式可能有所不同。如果值无法表示，则行为未定义。

返回值

如果值可以被表示，这些函数应返回转换后的值。

错误

未定义错误。

以下章节为参考信息。

示例

无。

应用程序用法

如果不知道数值是否在范围内，应使用 `strtol()` 或 `strtoll()`，因为 `atol()` 和 `atoll()` 不需要执行任何错误检查。

原理

无。

未来方向

无。

参见

- `strtol()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

添加了 `atoll()` 函数以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

应用了 SD5-XSH-ERN-61，更正了 `atoll()` 的描述。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0046 [892]。

1.9. atoll

SYNOPSIS

```
#include <stdlib.h>

long atol(const char *nptr);
long long atoll(const char *nptr);
```

DESCRIPTION

[选项开始] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 卷遵循 ISO C 标准。[选项结束]

除下面说明的情况外，调用 `atol(nptr)` 应等效于：

```
strtol(nptr, (char **)NULL, 10)
```

除下面说明的情况外，调用 `atoll(nptr)` 应等效于：

```
strtoll(nptr, (char **)NULL, 10)
```

错误处理可能有所不同。如果值无法表示，则行为未定义。

RETURN VALUE

这些函数返回转换后的值。

ERRORS

未定义错误。

EXAMPLES

未提供示例。

APPLICATION USAGE

未提供应用程序用法。

RATIONALE

未提供基本原理。

FUTURE DIRECTIONS

未提供未来方向。

SEE ALSO

`strtol()` , `strtoll()`

CHANGE HISTORY

已应用 SD5-XSH-ERN-61，更正了 `atoll()` 的 DESCRIPTION。

已应用 POSIX.1-2008 , Technical Corrigendum 2 , XSH/TC2-2008/0046 [892]。

信息性文本结束。

1.10. bsearch — 二分搜索已排序的表

概要

```
#include <stdlib.h>

void *bsearch(const void *key, const void *base, size_t nel,
              size_t width, int (*compar)(const void *, const v
```

描述

[CX] 本参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`bsearch()` 函数应搜索由 `nel` 个对象组成的数组，该数组的初始元素由 `base` 指向，寻找与 `key` 指向的对象匹配的元素。数组中每个元素的大小由 `width` 指定。如果 `nel` 参数值为零，则不应调用 `compar` 指向的比较函数，且不会找到匹配项。

`compar` 指向的比较函数应使用两个参数调用，这两个参数分别指向 `key` 对象和数组元素，顺序如此。

应用程序应确保 `compar` 指向的比较函数不改变数组的内容。实现可以在调用比较函数之间重新排序数组的元素，但不应改变任何单个元素的内容。

实现应确保第一个参数始终是指向键的指针。

当相同的对象（由 `width` 字节组成，无论它们在数组中的当前位置如何）被多次传递给比较函数时，结果应彼此一致。也就是说，相同的对象应始终以相同的方式与键进行比较。

应用程序应确保函数返回小于、等于或大于 0 的整数，如果 `key` 对象被认为分别小于、匹配或大于数组元素。应用程序应确保数组由所有比较小于键的对象、所有比较等于键的对象和所有比较大于键的对象组成，按此顺序排列。

返回值

`bsearch()` 函数应返回指向数组中匹配成员的指针，如果未找到匹配项则返回空指针。如果有两个或更多成员比较相等，则返回哪个成员是未指定的。

错误

未定义错误。

以下各节为提供信息的内容。

示例

下面的示例搜索一个表，该表包含指向由字符串及其长度组成的节点的指针。该表按照每个条目指向的节点中的字符串按字母顺序排序。

下面的代码片段读取字符串，要么找到相应的节点并打印出字符串及其长度，要么打印错误消息。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define TABSIZE      1000

struct node {                      /* 这些存储在表中。 */
    char *string;
    int length;
};

struct node table[TABSIZE];        /* 要搜索的表。 */

.

.

.

{

    struct node *node_ptr, node;
    /* 比较 2 个节点的例程。 */
    int node_compare(const void *, const void *);

.

.

.

    while (scanf("%ms", &node.string) != EOF) {
        node_ptr = (struct node *)bsearch((void *)(&node),
            (void *)table, TABSIZE,
            sizeof(struct node), node_compare);
        if (node_ptr != NULL) {
            (void)printf("string = %20s, length = %d\n",
                node_ptr->string, node_ptr->length);
        } else {
```

```
        (void)printf("not found: %s\n", node.string);
    }
    free(node.string);
}
/*
    这个例程基于字符串字段的字母顺序比较两个节点。
*/
int
node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
                   ((const struct node *)node2)->string);
}
```

应用程序用法

指向键和表基址元素的指针应为元素指针类型。

比较函数不需要比较每个字节，因此除了被比较的值之外，元素中可以包含任意数据。

在实践中，数组通常根据比较函数进行排序。

基本原理

要求比较函数的第二个参数（此后称为 `p`）是指向数组元素的指针，这意味着对于每次调用，以下所有表达式都不为零：

```
( (char *)p - (char *)base ) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```

未来方向

无。

另请参见

- [hcreate\(\)](#)

- `lsearch()`

- `qsort()`

- `tdelete()`

XBD `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，避免对应用程序要求使用"必须"一词。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/11，向描述中添加第一个非阴影段落的最后一句，以及以下三个段落。基本原理部分也已更新。这些更改是为了与 ISO C 标准保持一致。

Issue 7

示例部分已修订。

应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0051 [756]。

信息性文本结束。

1.11. `calloc` — 内存分配器

概要

```
#include <stdlib.h>

void *calloc(size_t nelem, size_t elsize);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`calloc()` 函数应为数组分配未使用的空间，该数组包含 `nelem` 个元素，每个元素的字节大小为 `elsize`。分配的空间应初始化为所有位均为 0。

通过连续调用 `calloc()` 分配的存储空间的顺序和连续性是未指定的。如果分配成功，返回的指针应适当对齐，以便可以将其赋值给具有基本对齐要求的任何类型对象的指针，然后用于访问在分配空间中的此类对象或此类对象的数组（直到空间被显式释放或重新分配）。每次这样的分配都应产生一个指向与其他任何对象不相交的对象的指针。返回的指针应指向分配空间的起始位置（最低字节地址）。如果无法分配空间，应返回空指针。如果请求的空间大小为 0，其行为是实现定义的：要么返回空指针，要么行为如同大小为某个非零值，但如果使用返回的指针访问对象，则行为未定义。

为确定数据竞争的存在性，`calloc()` 的行为应如同仅访问通过其参数可访问的内存位置，而不访问其他静态持续时间存储。但是，该函数可以可见地修改其分配的存储。对 `aligned_alloc()`、`calloc()`、`free()`、`malloc()`、[ADV] `posix_memalign()`、[CX] `reallocarray()` 和 `realloc()` 的调用，如果这些调用分配或释放特定的内存区域，则应在单一总顺序中发生（参见 4.15.1 内存排序），并且每个这样的释放调用应与该顺序中的下一次分配（如果有）同步。

返回值

成功完成时，`calloc()` 应返回指向分配空间的指针；如果 `nelem` 或 `elsize` 为 0，应用程序应确保不使用该指针访问对象。

否则，它应返回空指针 [CX] 并设置 `errno` 以指示错误。

错误

`calloc()` 函数在以下情况下可能失败：

- **[ENOMEM]** [CX] 可用内存不足，包括 `nelem` * `elsize` 会溢出的情况。

`calloc()` 函数在以下情况下可能失败：

- **[EINVAL]** [CX] `nelem` 或 `elsize` 为 0 且实现不支持大小为 0 的分配。
-

以下部分为参考信息。

示例

无。

应用程序用法

现在不再要求实现支持包含 `<malloc.h>`。

基本原理

参见 `malloc()` 的基本原理。

未来方向

无。

另请参见

- `aligned_alloc()`
- `free()`
- `malloc()`
- `realloc()`

XBD `<stdlib.h>`

更改历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 如果发生内存不足情况，设置 `errno` 和 [ENOMEM] 错误条件是强制性的。

Issue 7

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0053 [526]。

Issue 8

应用了 Austin Group 缺陷 374，更改了关于大小为 0 的分配的返回值和错误部分。

应用了 Austin Group 缺陷 1218，更改了 [ENOMEM] 错误。

应用了 Austin Group 缺陷 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group 缺陷 1387，更改了基本原理部分。

1.12. clearerr

SYNOPSIS

```
#include <stdio.h>

void clearerr(FILE *stream);
```

DESCRIPTION

本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`clearerr()` 函数应清除 `stream` 参数所指向流的文件结束指示器和错误指示器。

如果 `stream` 参数有效, `clearerr()` 函数不应更改 `errno` 的设置。

RETURN VALUE

`clearerr()` 函数不返回任何值。

ERRORS

未定义任何错误。

以下部分为提供参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

XBD [`<stdio.h>`](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008 Technical Corrigendum 1, XSH/TC1-2008/0057 [401]。

1.13. `clock_getres`, `clock_gettime`, `clock_settime` — 时钟和定时器函数

概述

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp)
```

描述

`clock_getres()` 函数应返回任何时钟的分辨率。时钟分辨率是由实现定义的，不能由进程设置。如果参数 `res` 不为 NULL，指定时钟的分辨率应存储在 `res` 指向的位置。如果 `res` 为 NULL，则不返回时钟分辨率。如果 `clock_settime()` 的时间参数不是 `res` 的倍数，则该值被截断为 `res` 的倍数。

`clock_gettime()` 函数应返回指定时钟 `clock_id` 的当前值 `tp`。

`clock_settime()` 函数应将指定时钟 `clock_id` 设置为 `tp` 指定的值。介于指定时钟分辨率的两个连续非负整数倍数之间的时间值应被向下截断为较小的分辨率倍数。

时钟可以是系统范围的（即对所有进程可见）或每进程的（仅在进程内有意义的时间测量）。所有实现都应支持 `<time.h>` 中定义的 `clock_id` `CLOCK_REALTIME`。此时钟表示测量系统实时时间的时钟。对于此时钟，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示自纪元以来的时间量（以秒和纳秒为单位）。实现也可以支持额外的时钟。这些时钟的时间值解释是未指定的。

如果通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值，时钟的新值应用于确定基于 `CLOCK_REALTIME` 时钟的绝对时间服务的到期时间。这适用于已启动的绝对定时器到期的时间。如果在调用此类时间服务时请求的绝对时间早于时钟的新值，则时间服务应立即到期，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值对基于此时钟等待相对时间服务的阻塞线程（包括 `nanosleep()` 和 `thrd_sleep()` 函数）没

有影响；对基于此时钟的相对定时器的到期也没有影响。因此，这些时间服务应在请求的相对间隔过去时到期，与时钟的新值或旧值无关。

所有实现都应支持 `<time.h>` 中定义的 `clock_id` `CLOCK_MONOTONIC`。此时钟表示系统的单调时钟。对于此时钟，`clock_gettime()` 返回的值表示自过去某个未指定点（例如，系统启动时间或纪元）以来的时间量（以秒和纳秒为单位）。此点在系统启动时间后不会改变。`CLOCK_MONOTONIC` 时钟的值不能通过 `clock_settime()` 设置。如果使用 `CLOCK_MONOTONIC` 的 `clock_id` 参数调用此函数，则该函数应失败。

通过 `clock_settime()` 设置时钟对除 `CLOCK_REALTIME` 之外的其他时钟关联的已启动每进程定时器的影响是由实现定义的。

如果通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值，时钟的新值应用于确定系统应唤醒基于 `CLOCK_REALTIME` 时钟阻塞在绝对 `clock_nanosleep()` 调用上的线程的时间。如果在调用此类时间服务时请求的绝对时间早于时钟的新值，则调用应立即返回，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值对任何阻塞在相对 `clock_nanosleep()` 调用上的线程没有影响。因此，调用应在请求的相对间隔过去时返回，与时钟的新值或旧值无关。

设置特定时钟的适当特权是由实现定义的。

CPU 时间时钟支持（选项：CPT）

如果定义了 `_POSIX_CPUTIME`，实现应支持通过调用 `clock_getcpu_clockid()` 获得的时钟 ID 值，这些值表示给定进程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_PROCESS_CPUTIME_ID`，该值在调用 `clock_*` 或 `timer_*` 函数之一时表示调用进程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联的进程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对间歇服务器调度策略的行为没有影响。

线程 CPU 时间时钟支持（选项：TCT）

如果定义了 `_POSIX_THREAD_CPUTIME`，实现应支持通过调用 `pthread_getcpu_clockid()` 获得的时钟 ID 值，这些值表示给定线程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_THREAD_CPUTIME_ID`，该值在调用 `clock_*` 或 `timer_*` 函数之一时表示调用线程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联

的线程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对间歇服务器调度策略的行为没有影响。

返回值

返回值为 0 应表示调用成功。返回值为 -1 应表示发生了错误，并且 `errno` 应被设置为指示错误。

错误

`clock_getres()`、`clock_gettime()` 和 `clock_settime()` 函数在以下情况下应失败：

- **[EINVAL]**

`clock_id` 参数未指定已知时钟。

`clock_gettime()` 函数在以下情况下应失败：

- **[EOVERFLOW]**

秒数不适合 `time_t` 类型的对象。

`clock_settime()` 函数在以下情况下应失败：

- **[EINVAL]**

`clock_settime()` 的 `tp` 参数超出了给定时间 ID 的范围。

- **[EINVAL]**

`tp` 参数指定的纳秒值小于零或大于等于 1000 百万。

- **[EINVAL]**

`clock_id` 参数的值是 `CLOCK_MONOTONIC`。

`clock_settime()` 函数在以下情况下可能失败：

- **[EPERM]**

请求进程没有设置指定时钟的适当特权。

应用程序用法

请注意，单调时钟的绝对值是无意义的（因为其原点是任意的），因此无需设置它。此外，实时应用程序可以依赖这样一个事实：此时钟的值永远不会被设

置，因此用此时钟测量的时间间隔不会受到 `clock_settime()` 调用的影响。

示例

无。

原理

无。

未来方向

无。

另请参阅

- 调度策略
- `clock_getcpu_clockid()`
- `clock_nanosleep()`
- `ctime()`
- `mq_receive()`
- `mq_send()`
- `nanosleep()`
- `pthread_mutex_clockclock()`
- `sem_clockwait()`
- `thrd_sleep()`
- `time()`
- `timer_create()`
- `timer_getoverrun()`

XBD `<time.h>`

变更历史

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

- [ENOSYS] 错误条件已被删除，因为如果实现不支持定时器选项，则不需要提供存根。
- 添加了应用程序用法部分。
- 添加了重置时钟分辨率影响的说明。
- 为与 IEEE Std 1003.1d-1999 对齐而添加了 CPU 时间时钟和 `clock_getcpu_clockid()` 函数。
- 为与 IEEE Std 1003.1j-2000 对齐的更改：
- 更新了描述，如下：
 - 指定了 `clock_gettime()` 为 CLOCK_MONOTONIC 返回的值。
 - 指定了 `clock_settime()` 函数对 CLOCK_MONOTONIC 失败。
 - 指定了 `clock_settime()` 对 `clock_nanosleep()` 函数相对于 CLOCK_REALTIME 的影响。
- 在错误部分添加了 [EINVAL] 错误，表示 `clock_settime()` 对 CLOCK_MONOTONIC 失败。
- 应用程序用法部分指出，CLOCK_MONOTONIC 时钟无需也不应由 `clock_settime()` 设置，因为 CLOCK_MONOTONIC 时钟的绝对值是无意义的。
- 将 `clock_nanosleep()`、`mq_timedreceive()`、`mq_timedsend()`、`pthread_mutex_timedlock()`、`sem_timedwait()`、`timer_create()` 和 `timer_settime()` 函数添加到另请参阅部分。

Issue 7

- 与时钟选择选项相关的功能被移至基础。
- `clock_getres()`、`clock_gettime()` 和 `clock_settime()` 函数从 定时器选项移至基础。

- 应用了 POSIX.1-2008 技术勘误 1, XSH/TC1-2008/0058 [106]。

Issue 8

- 应用了 Austin Group 缺陷 1302, 将"the `nanosleep()` function"更改
为"the `nanosleep()` and `thrd_sleep()` functions"。
 - 应用了 Austin Group 缺陷 1346, 要求支持单调时钟。
-

1.14. `clock_getres`, `clock_gettime`, `clock_settime`

SYNOPSIS

```
[CX] #include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp
```

DESCRIPTION

`clock_getres()` 函数应返回任何时钟的分辨率。时钟分辨率由实现定义，不能由进程设置。如果参数 `res` 不为 NULL，指定时钟的分辨率应存储在 `res` 指向的位置。如果 `res` 为 NULL，则不返回时钟分辨率。如果 `clock_settime()` 的 `time` 参数不是 `res` 的倍数，则该值将被截断为 `res` 的倍数。

`clock_gettime()` 函数应返回指定时钟 `clock_id` 的当前值 `tp`。

`clock_settime()` 函数应将指定时钟 `clock_id` 设置为由 `tp` 指定的值。介于指定时钟分辨率的两个连续非负整数倍数之间的时间值应被截断为较小的分辨率倍数。

时钟可以是系统范围的（即对所有进程可见）或每进程的（仅在进程内有意义的时间测量）。所有实现都应支持 `<time.h>` 中定义的 `clock_id` `CLOCK_REALTIME`。此时钟表示测量系统实时时间的时钟。对于此时钟，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示自纪元以来的时间量（以秒和纳秒为单位）。实现也可以支持其他时钟。这些时钟的时间值解释未指定。

如果 `CLOCK_REALTIME` 时钟的值通过 `clock_settime()` 设置，时钟的新值应用于确定基于 `CLOCK_REALTIME` 时钟的绝对时间服务的到期时间。这适用于已启动的绝对计时器的到期时间。如果在调用此类时间服务时请求的绝对时间在时钟的新值之前，则时间服务应立即到期，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值对基于此时钟的相对时间服务阻塞等待的线程没有影响，包括 `nanosleep()` 和 `thrd_sleep()`

函数；对基于此时钟的相对计时器的到期也没有影响。因此，这些时间服务应在请求的相对间隔到期时到期，独立于时钟的新值或旧值。

所有实现都应支持 `<time.h>` 中定义的 `clock_id` `CLOCK_MONOTONIC`。此时钟表示系统的单调时钟。对于此时钟，`clock_gettime()` 返回的值表示自过去某个未指定点（例如，系统启动时间或纪元）以来的时间量（以秒和纳秒为单位）。此点在系统启动时间后不会更改。`CLOCK_MONOTONIC` 时钟的值不能通过 `clock_settime()` 设置。如果使用 `CLOCK_MONOTONIC` 的 `clock_id` 参数调用此函数，则该函数应失败。

通过 `clock_settime()` 设置时钟对与除 `CLOCK_REALTIME` 之外的时钟关联的已启动每进程计时器的影响是实现定义的。

如果 `CLOCK_REALTIME` 时钟的值通过 `clock_settime()` 设置，时钟的新值应用于确定系统应唤醒在基于 `CLOCK_REALTIME` 时钟的绝对 `clock_nanosleep()` 调用上阻塞的线程的时间。如果在调用此类时间服务时请求的绝对时间在时钟的新值之前，则调用应立即返回，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值对在相对 `clock_nanosleep()` 调用上阻塞的任何线程没有影响。因此，调用应在请求的相对间隔到期时返回，独立于时钟的新值或旧值。

设置特定时钟的适当权限是实现定义的。

[CPT] 如果定义了 `_POSIX_CPUTIME`，实现应支持通过调用 `clock_getcpu_clockid()` 获得的时钟 ID 值，这些值表示给定进程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_PROCESS_CPUTIME_ID`，当调用 `clock_*` 或 `timer_*` 函数之一时，它表示调用进程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联的进程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对偶发服务器调度策略的行为没有影响（参见[调度策略]）。

[TCT] 如果定义了 `_POSIX_THREAD_CPUTIME`，实现应支持通过调用 `pthread_getcpu_clockid()` 获得的时钟 ID 值，这些值表示给定线程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_THREAD_CPUTIME_ID`，当调用 `clock_*` 或 `timer_*` 函数之一时，它表示调用线程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联的线程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对偶发服务器调度策略的行为没有影响（参见[调度策略]）。

RETURN VALUE

返回值为 0 表示调用成功。返回值为 -1 表示发生错误，并且应设置 `errno` 以指示错误。

ERRORS

`clock_getres()`、`clock_gettime()` 和 `clock_settime()` 函数在以下情况下应失败：

- **[EINVAL]** `clock_id` 参数未指定已知时钟。

`clock_gettime()` 函数在以下情况下应失败：

- **[EOVERFLOW]** 秒数无法放入 `time_t` 类型的对象中。

`clock_settime()` 函数在以下情况下应失败：

- **[EINVAL]** `clock_settime()` 的 `tp` 参数超出了给定时钟 ID 的范围。
- **[EINVAL]** `tp` 参数指定的纳秒值小于零或大于等于 1000 百万。
- **[EINVAL]** `clock_id` 参数的值为 `CLOCK_MONOTONIC`。

`clock_settime()` 函数在以下情况下可能失败：

- **[EPERM]** 请求进程没有设置指定期钟的适当权限。

EXAMPLES

无。

APPLICATION USAGE

注意单调时钟的绝对值没有意义（因为其原点是任意的），因此无需设置它。此外，实时应用程序可以依赖此时钟的值永远不会被设置这一事实，因此使用此时钟测量的时间间隔不会受到对 `clock_settime()` 调用的影响。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

[调度策略], `clock_getcpu_clockid()`, `clock_nanosleep()`, `ctime()`,
`mq_receive()`, `mq_send()`, `nanosleep()`,
`pthread_mutex_clocklock()`, `sem_clockwait()`, `thrd_sleep()`,
`time()`, `timer_create()`, `timer_getoverrun()`

XBD `<time.h>`

CHANGE HISTORY

首次在 Issue 5 中发布。为与 POSIX 实时扩展对齐而包含。

Issue 6

如果实现不支持计时器选项，则不需要提供存根，[ENOSYS] 错误条件已被删除。

添加了 APPLICATION USAGE 部分。

为与 IEEE P1003.1a 草案标准对齐进行了以下更改：

- 添加了重置时钟分辨率影响的说明。
- 为与 IEEE Std 1003.1d-1999 对齐，添加了 CPU 时间时钟和 `clock_getcpu_clockid()` 函数。

为与 IEEE Std 1003.1j-2000 对齐添加了以下更改：

- DESCRIPTION 更新如下：
- 指定了 CLOCK_MONOTONIC 的 `clock_gettime()` 返回值。
- 指定了 CLOCK_MONOTONIC 的 `clock_settime()` 函数失败。
- 指定了 `clock_settime()` 对 CLOCK_REALTIME 的 `clock_nanosleep()` 函数的影响。
- 在 ERRORS 部分添加了 [EINVAL] 错误，指示 `clock_settime()` 对 CLOCK_MONOTONIC 失败。

- APPLICATION USAGE 部分指出 CLOCK_MONOTONIC 时钟不需要也不应由 `clock_settime()` 设置, 因为 CLOCK_MONOTONIC 时钟的绝对值没有意义。
- `clock_nanosleep()`、`mq_timedreceive()`、`mq_timedsend()`、`pthread_mutex_timedlock()`、`sem_timedwait()`、`timer_create()` 和 `timer_settime()` 函数被添加到 SEE ALSO 部分。

Issue 7

与时钟选择选项相关的功能被移至基础。

`clock_getres()`、`clock_gettime()` 和 `clock_settime()` 函数从计时器选项移至基础。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0058 [106]。

Issue 8

应用 Austin Group Defect 1302, 将"the `nanosleep()` function"更改为"the `nanosleep()` and `thrd_sleep()` functions"。

应用 Austin Group Defect 1346, 要求支持单调时钟。

1.15. `clock_nanosleep`

SYNOPSIS

```
#include <time.h>

int clock_nanosleep(clockid_t clock_id, int flags,
                     const struct timespec *rqtp, struct timespec
```

DESCRIPTION

如果在 `flags` 参数中未设置 `TIMER_ABSTIME` 标志，`clock_nanosleep()` 函数将使当前线程暂停执行，直到 `rqtp` 参数指定的时间间隔已经过去，或者向调用线程传递一个信号且其动作是调用信号捕获函数，或者进程终止。用于测量时间的时钟应该是 `clock_id` 指定的时钟。

如果在 `flags` 参数中设置了 `TIMER_ABSTIME` 标志，`clock_nanosleep()` 函数将使当前线程暂停执行，直到 `clock_id` 指定的时钟的时间值达到 `rqtp` 参数指定的绝对时间，或者向调用线程传递一个信号且其动作是调用信号捕获函数，或者进程终止。如果在调用时，`rqtp` 指定的时间值小于或等于指定时钟的时间值，那么 `clock_nanosleep()` 应立即返回，调用进程不应被暂停。

此函数导致的暂停时间可能比请求的时间更长，因为参数值会向上舍入为休眠分辨率的整数倍，或者因为系统对其他活动的调度。但是，除了被信号中断的情况外，相对 `clock_nanosleep()` 函数（即未设置 `TIMER_ABSTIME` 标志）的暂停时间不应小于 `rqtp` 指定的时间间隔，这以相应时钟的测量为准。绝对 `clock_nanosleep()` 函数（即设置了 `TIMER_ABSTIME` 标志）的暂停应至少持续到相应时钟的值达到 `rqtp` 指定的绝对时间为止，除了被信号中断的情况。

使用 `clock_nanosleep()` 函数对任何信号的动作或阻塞没有影响。

如果 `clock_id` 参数指代调用线程的 CPU 时间时钟，`clock_nanosleep()` 函数应该失败。是否允许其他 CPU 时间时钟的 `clock_id` 值是未指定的。

RETURN VALUE

如果 `clock_nanosleep()` 函数因为请求的时间已经过去而返回，其返回值应该为零。

如果 `clock_nanosleep()` 函数因为被信号中断而返回，它应该返回相应的错误值。对于相对 `clock_nanosleep()` 函数，如果 `rmtp` 参数非 NULL，它引用的 `timespec` 结构应该被更新为包含间隔内剩余的时间量（请求时间减去实际睡眠时间）。`rqtp` 和 `rmtp` 参数可以指向同一个对象。如果 `rmtp` 参数为 NULL，则不返回剩余时间。绝对 `clock_nanosleep()` 函数对 `rmtp` 引用的结构没有影响。

如果 `clock_nanosleep()` 失败，它应该返回相应的错误值。

ERRORS

`clock_nanosleep()` 函数在以下情况下应该失败：

- **[EINTR]**

`clock_nanosleep()` 函数被信号中断。

- **[EINVAL]**

`rqtp` 参数指定的纳秒值小于零或大于等于 1000 百万；或者在 `flags` 中指定了 `TIMER_ABSTIME` 标志，且 `rqtp` 参数在 `clock_id` 指定的时钟范围之外；或者 `clock_id` 参数没有指定已知时钟，或者指定了调用线程的 CPU 时间时钟。

- **[ENOTSUP]**

`clock_id` 参数指定了一个不支持 `clock_nanosleep()` 的时钟，例如 CPU 时间时钟。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

使用未在 `flags` 参数中设置 `TIMER_ABSTIME` 值且 `clock_id` 为 `CLOCK_REALTIME` 的 `clock_nanosleep()` 调用等同于使用相同 `rqtp` 和 `rmtp` 参数的 `nanosleep()` 调用。

RATIONALE

`nanosleep()` 函数指定使用系统范围的时钟 `CLOCK_REALTIME` 来测量此时间服务的经过时间。然而，随着单调时钟 `CLOCK_MONOTONIC` 的引入，需要一个新的相对休眠函数来允许应用程序利用这种时钟的特殊特性。

在许多应用程序中，进程需要以周期方式多次暂停然后激活；例如，轮询非中断设备的状态或刷新显示设备。对于这些情况，已知无法使用相对 `sleep()` 或 `nanosleep()` 函数调用实现精确的周期性激活。例如，假设一个周期性进程在时间 T_0 激活，执行一段时间，然后想要暂停自身直到时间 T_0+T ，周期为 T 。如果此进程想要使用 `nanosleep()` 函数，它必须首先调用 `clock_gettime()` 获取当前时间，然后计算当前时间与 T_0+T 之间的差异，最后使用计算出的间隔调用 `nanosleep()`。然而，进程可能在两个函数调用之间被不同的进程抢占，在这种情况下计算的间隔会是错误的；进程会比期望的更晚唤醒。使用绝对 `clock_nanosleep()` 函数不会出现这个问题，因为只需要一个函数调用来暂停进程直到期望的时间。但在其他情况下，需要相对休眠，这就是为什么需要这两种功能。

虽然可以使用定时器接口实现周期性进程，但这种实现需要使用信号，并保留一些信号编号。在这方面，在 POSIX.1-2024 中包含 `clock_nanosleep()` 函数的绝对版本的原因与包含相对 `nanosleep()` 的原因相同。

也可以使用 `pthread_cond_timedwait()` 实现精确的周期性进程，其中指定一个绝对超时，如果涉及的条件变量从未被发信号，则该超时生效。然而，使用这种接口是不自然的，并且涉及对互斥锁和条件变量执行其他操作，这意味着不必要的开销。此外，在不支持线程的实现中，`pthread_cond_timedwait()` 不可用。

虽然新的高精度休眠服务的相对版本和绝对版本的接口是同一个 `clock_nanosleep()` 函数，但 `rmtp` 参数仅在相对休眠中使用。在相对 `clock_nanosleep()` 函数中需要此参数来在函数被信号中断时重新发出函数调用，但在绝对 `clock_nanosleep()` 函数调用中不需要此参数；如果调用被信号中断，绝对 `clock_nanosleep()` 函数可以使用被中断调用中使用的相同 `rqtp` 参数再次调用。

FUTURE DIRECTIONS

无。

SEE ALSO

- `clock_getres()`
- `nanosleep()`
- `pthread_cond_clockwait()`
- `sleep()`

XBD `<time.h>`

CHANGE HISTORY

首次发布于 Issue 6。源自 IEEE Std 1003.1j-2000。

Issue 7

`clock_nanosleep()` 函数从时钟选择选项移动到基础规范。

应用 POSIX.1-2008 Technical Corrigendum 2, XSH/TC2-2008/0068 [909]。

1.16. `clock_settime`

概要

```
#include <time.h>

int clock_getres(clockid_t clock_id, struct timespec *res);
int clock_gettime(clockid_t clock_id, struct timespec *tp);
int clock_settime(clockid_t clock_id, const struct timespec *tp)
```

描述

`clock_getres()` 函数应返回任何时钟的分辨率。时钟分辨率由实现定义，进程无法设置。如果参数 `res` 不为 `NULL`，指定时钟的分辨率应存储在 `res` 指向的位置。如果 `res` 为 `NULL`，则不返回时钟分辨率。如果 `clock_settime()` 的 `time` 参数不是 `res` 的倍数，则该值将被截断为 `res` 的倍数。

`clock_gettime()` 函数应返回指定时钟 `clock_id` 的当前值 `tp`。

`clock_settime()` 函数应将指定时钟 `clock_id` 设置为 `tp` 指定的值。介于指定时钟分辨率的两个连续非负整数倍之间的时间值应向下截断为较小的分辨率倍数。

时钟可以是系统范围的（即对所有进程可见）或每进程的（仅在进程内有意义的时间测量）。所有实现都应支持 `<time.h>` 中定义的 `clock_id` `CLOCK_REALTIME`。该时钟表示测量系统实时时间的时钟。对于此时钟，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示自纪元以来的时间量（以秒和纳秒为单位）。实现也可以支持额外的时钟。这些时钟的时间值解释是未指定的。

`CLOCK_REALTIME` 行为

如果通过 `clock_settime()` 设置 `CLOCK_REALTIME` 时钟的值，则应使用时钟的新值来确定基于 `CLOCK_REALTIME` 时钟的绝对时间服务的到期时间。这适用于已启动的绝对定时器到期的时间。如果在调用此类时间服务时请求的绝对时间在时钟的新值之前，则该时间服务应立即到期，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 CLOCK_REALTIME 时钟的值对基于此时钟等待相对时间服务的阻塞线程没有影响，包括 `nanosleep()` 和 `thrd_sleep()` 函数；对基于此时钟的相对定时器的到期也没有影响。因此，这些时间服务应在请求的相对间隔过去时到期，与时钟的新值或旧值无关。

CLOCK_MONOTONIC 行为

所有实现都应支持 `<time.h>` 中定义的 `clock_id` CLOCK_MONOTONIC。该时钟表示系统的单调时钟。对于此时钟，`clock_gettime()` 返回的值表示自过去某个未指定点（例如，系统启动时间或纪元）以来的时间量（以秒和纳秒为单位）。此点在系统启动时间后不会改变。CLOCK_MONOTONIC 时钟的值无法通过 `clock_settime()` 设置。如果使用 CLOCK_MONOTONIC 的 `clock_id` 参数调用此函数，则该函数应失败。

clock_nanosleep 交互

如果通过 `clock_settime()` 设置 CLOCK_REALTIME 时钟的值，则应使用时钟的新值来确定系统唤醒基于 CLOCK_REALTIME 时钟阻塞在绝对 `clock_nanosleep()` 调用上的线程的时间。如果在调用此类时间服务时请求的绝对时间在时钟的新值之前，则该调用应立即返回，就像时钟已正常达到请求的时间一样。

通过 `clock_settime()` 设置 CLOCK_REALTIME 时钟的值对任何阻塞在相对 `clock_nanosleep()` 调用上的线程没有影响。因此，该调用应在请求的相对间隔过去时返回，与时钟的新值或旧值无关。

定时器效果

通过 `clock_settime()` 设置时钟对与除 CLOCK_REALTIME 之外的时钟关联的已启动每进程定时器的效果是实现定义的。

权限

设置特定时钟的适当权限是实现定义的。

CPU 时间时钟 [CPT]

如果定义了 `_POSIX_CPUTIME`，实现应支持通过调用 `clock_getcpu_clockid()` 获得的时钟 ID 值，这些值表示给定进程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_PROCESS_CPUTIME_ID`，该值在调用 `clock_*` 或 `timer_*` 函数时使用。

数之一时表示调用进程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联的进程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对零星服务器调度策略的行为没有影响。

线程 CPU 时间时钟 [TCT]

如果定义了 `_POSIX_THREAD_CPUTIME`，实现应支持通过调用 `pthread_getcpuclockid()` 获得的时钟 ID 值，这些值表示给定线程的 CPU 时间时钟。实现还应支持特殊的 `clockid_t` 值 `CLOCK_THREAD_CPUTIME_ID`，该值在调用 `clock_*` 或 `timer_*` 函数之一时表示调用线程的 CPU 时间时钟。对于这些时钟 ID，`clock_gettime()` 返回的值和 `clock_settime()` 指定的值表示与时钟关联的线程的执行时间量。通过 `clock_settime()` 更改 CPU 时间时钟的值对零星服务器调度策略的行为没有影响。

返回值

成功完成后，`clock_getres()`、`clock_gettime()` 和 `clock_settime()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

错误

这些函数可能在以下情况下失败：

- **EINVAL:** `clock_id` 参数未指定已知时钟，或 `tp` 参数超出指定时钟的范围。

`clock_settime()` 函数可能在以下情况下失败：

- **EPERM:** 请求进程没有设置指定时钟的适当权限。
- **EINVAL:** `clock_id` 参数是 `CLOCK_MONOTONIC`。

这些函数可能在以下情况下失败：

- **EOVERFLOW:** 秒数无法放入 32 位有符号整数。

示例

本规范中未提供示例。

应用程序用法

以下部分是参考信息。

基本原理

未提供。

未来方向

未提供。

另请参阅

- `clock_getcpu_clockid()`
- `clock_nanosleep()`
- `nanosleep()`
- `pthread_getcpu_clockid()`
- `thrd_sleep()`
- `<time.h>`

XSH

第 7 版

以下部分是参考信息。

1.17. close, posix_close

SYNOPSIS (概要)

```
#include <unistd.h>

int close(int fildes);
int posix_close(int fildes, int flag);
```

DESCRIPTION (描述)

`close()` 函数应释放由 `fildes` 指示的文件描述符。释放意味着使该文件描述符可通过后续的 `open()` 调用或其他分配文件描述符的函数返回。调用进程在该文件描述符关联的文件上拥有的所有进程拥有的文件锁都应被解锁。

如果 `close()` 被一个要捕获的信号中断，那么它是不确定的：是返回 -1 并将 `errno` 设置为 [EINTR] 且 `fildes` 保持打开状态，还是返回 -1 并将 `errno` 设置为 [EINPROGRESS] 且 `fildes` 被关闭，或是返回 0 表示成功完成；除非 `POSIX_CLOSE_RESTART` 被定义为 0，在这种情况下，返回 -1 并将 `errno` 设置为 [EINTR] 且 `fildes` 保持打开状态的选项不会发生。如果 `close()` 返回 -1 并将 `errno` 设置为 [EINTR]，那么除了 `close()` 或 `posix_close()` 之外，`fildes` 是否可以后续传递给任何函数而不出错是不确定的。对于所有其他错误情况（除了 [EBADF]，其中 `fildes` 是无效的），`fildes` 应被关闭。如果即使关闭操作未完成 `fildes` 也被关闭了，关闭操作应继续异步进行，进程应没有进一步跟踪关闭操作完成或最终状态的能力。

当与管道或 FIFO 特殊文件关联的所有文件描述符都被关闭时，管道或 FIFO 中剩余的任何数据都应被丢弃。

当与打开文件描述关联的所有文件描述符都被关闭时，打开文件描述应被释放。

如果文件的链接计数为 0，当与文件关联的所有文件描述符都被关闭时，文件占用的空间应被释放，文件应不再可访问。

[XSI] 如果 `fildes` 指的是伪终端的管理方端，并且这是最后一次关闭，应向控制进程发送 SIGHUP 信号（如果有的话），对于该进程，伪终端的附属方端是控制终端。关闭伪终端的管理方端是否会刷新所有排队的输入和输出是不确定的。

当调用 `close()` 时, 如果存在针对 `fildes` 的未完成的可取消异步 I/O 操作, 该 I/O 操作可能会被取消。未被取消的 I/O 操作会像 `close()` 操作尚未发生一样完成。所有未被取消的操作应像 `close()` 阻塞直到操作完成为止一样完成。`close()` 操作本身不需要阻塞等待此类 I/O 完成。任何 I/O 操作是否被取消, 以及 `close()` 时哪些 I/O 操作可能被取消, 是由实现定义的。

如果在最后一次关闭时内存映射文件 [SHM] 或共享内存对象仍然被引用 (即进程已映射它), 那么内存对象的全部内容应持续存在, 直到内存对象变为未被引用。如果这是内存映射文件 [SHM] 或共享内存对象的最后一次关闭, 并且关闭导致内存对象变为未被引用, 且内存对象已被取消链接, 那么内存对象应被移除。

当与套接字关联的所有文件描述符都被关闭时, 套接字应被销毁。如果套接字处于连接模式, 并且为套接字设置了 SO_LINGER 选项且具有非零延迟时间, 并且套接字有未传输的数据, 那么 `close()` 应阻塞最多当前延迟间隔, 直到所有数据都被传输。

`posix_close()` 函数应等同于 `close()` 函数, 除了基于 `flag` 参数的如下所述的修改。如果 `flag` 为 0, 那么 `posix_close()` 不应返回 -1 并将 `errno` 设置为 [EINTR], 这意味着 `fildes` 将始终被关闭 (除了 [EBADF], 其中 `fildes` 是无效的)。如果 `flag` 包含 POSIX_CLOSE_RESTART 且 POSIX_CLOSE_RESTART 被定义为非零值, 并且 `posix_close()` 被一个要捕获的信号中断, 那么 `posix_close()` 可能返回 -1 并将 `errno` 设置为 [EINTR], 在这种情况下 `fildes` 应保持打开状态; 但是, 除了 `close()` 或 `posix_close()` 之外, `fildes` 是否可以后续传递给任何函数而不出错是不确定的。如果 `flag` 无效, `posix_close()` 可能失败并将 `errno` 设置为 [EINVAL], 但在其他情况下应像 `flag` 为 0 一样行为并关闭 `fildes`。

RETURN VALUE (返回值)

成功完成后, 应返回 0; 否则, 应返回 -1 并设置 `errno` 以指示错误。

ERRORS (错误)

`close()` 和 `posix_close()` 函数在以下情况下应失败:

- [EBADF]
`fildes` 参数不是一个打开的文件描述符。

- [EINPROGRESS]
函数被信号中断, `fildes` 被关闭但关闭操作正在异步继续。

`close()` 和 `posix_close()` 函数在以下情况下可能失败：

- **[EINTR]**

函数被信号中断，`POSIX_CLOSE_RESTART` 被定义为非零值，并且（在 `posix_close()` 的情况下）`flag` 参数包含 `POSIX_CLOSE_RESTART`，在这种情况下 `fildes` 仍然打开。

- **[EIO]**

在从文件系统读取或向文件系统写入时发生 I/O 错误。

`posix_close()` 函数在以下情况下可能失败：

- **[EINVAL]**

`flag` 参数的值无效。

`close()` 和 `posix_close()` 函数不应返回 `[EAGAIN]` 或 `[EWOULDBLOCK]` 错误。如果 `POSIX_CLOSE_RESTART` 为零，`close()` 函数不应返回 `[EINTR]` 错误。`posix_close()` 函数不应返回 `[EINTR]` 错误，除非 `flag` 包含非零的 `POSIX_CLOSE_RESTART`。

以下部分为参考信息。

EXAMPLES (示例)

重新分配文件描述符

以下示例关闭当前进程与标准输出关联的文件描述符，将标准输出重新分配给一个新的文件描述符，然后关闭原始文件描述符进行清理。此示例假设文件描述符 0（标准输入的描述符）没有被关闭。

```
#include <unistd.h>
...
int pfd;
...
close(1);
dup(pfd);
close(pfd);
...
```

顺便说一下，这完全可以使用以下方式实现：

```
dup2(pfd, 1);
close(pfd);
```

关闭文件描述符

在以下示例中，`close()` 用于在尝试将文件描述符与流关联失败后关闭该文件描述符。

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
FILE *fpfd;
...
if ((fpfd = fdopen (pfd, "w")) == NULL) {
    close(pfd);
    unlink(LOCKFILE);
    exit(1);
}
...
```

APPLICATION USAGE (应用程序使用)

使用了 `stdio` 例程 `fopen()` 打开文件的应用程序应使用相应的 `fclose()` 例程而不是 `close()`。一旦文件被关闭，文件描述符就不再存在，因为与其对应的整数不再引用文件。

实现可能使用必须继承到子进程的文件描述符，以使子进程保持符合规范，例如用于消息目录或跟踪目的。因此，对任意整数调用 `close()` 的应用程序有不符合规范行为的风险，`close()` 只能可移植地用于应用程序通过显式操作获得的文件描述符值，以及与标准文件流对应的三个文件描述符。在多线程父应用程序中，在 `fork()` 之后和 `exec` 调用之前在循环中调用 `close()` 以避免将意外文件描述符泄漏到子进程的竞争条件的做法因此是不安全的，竞争应通过使用 `FD_CLOEXEC` 位设置打开所有文件描述符来对抗，除非文件描述符旨在跨 `exec` 继承。

在文件描述符 `STDIN_FILENO`、`STDOUT_FILENO` 或 `STDERR_FILENO` 上使用 `close()` 应立即跟随重新打开这些文件描述符的操作。如果这些文件描述符中的任何一个保持关闭状态（例如，来自 `perror()` 的 `[EBADF]` 错误），或者如

果应用程序后面不相关的 `open()` 或类似调用意外地将文件分配给这些众所周知的文件描述符之一，将导致意外行为。此外，`close()` 后跟重新打开操作（例如，`open()`、`dup()` 等）不是原子操作；应使用 `dup2()` 来更改标准文件描述符。

RATIONALE (原理)

应避免使用可中断设备关闭例程，以避免与文件描述符的隐式关闭（如通过 `exec`、进程终止或 `dup2()`）相关的问题。POSIX.1-2024 的这一卷仅通过为具有非零 `POSIX_CLOSE_RESTART` 的 `close()` 和 `posix_close()` 指定 [EINTR] 错误条件来允许此类行为，以便为应用程序提供可移植的方式，在接收到中断后恢复等待与关闭操作相关的事件（例如，磁带机倒带）。本标准还允许实现如果选择不提供重新启动中断关闭操作的方式，则将 `POSIX_CLOSE_RESTART` 定义为 0。尽管在 [EINTR] 时文件描述符保持打开状态，但它可能不再可用——也就是说，将其传递给除 `close()` 或 `posix_close()` 之外的任何函数可能导致诸如 [EIO] 之类的错误。如果应用程序必须保证数据不会丢失，建议应用程序在关闭操作之前使用 `fsync()` 或 `fdatasync()`，而不是让关闭操作处理待处理的 I/O 并承担中断的风险。

本标准的早期版本在 [EINTR] 和 [EIO] 等错误之后将 `fildes` 的状态保留为未指定；实现在 `close()` 在 [EINTR] 后是否保持 `fildes` 打开方面存在差异。一旦引入线程，这是不令人满意的，因为多线程应用程序需要知道 `fildes` 是否已被关闭。应用程序不能盲目地再次调用 `close()`，因为如果 `fildes` 被第一次调用关闭，另一个线程可能已被分配了一个与 `fildes` 相同值的文件描述符，该文件描述符绝不能被第一个线程关闭。另一方面，从不重试 `close()` 的替代方案将在 `close()` 未关闭 `fildes` 的实现中导致文件描述符泄漏，尽管如果进程即将退出或文件描述符被标记为 `FD_CLOEXEC` 且进程即将被 `exec` 替换，这种泄漏可能是无害的。本标准引入了带有 `flag` 参数的 `posix_close()`，它允许在两种可能的错误行为之间进行选择，并将 `close()` 实现两种行为中的哪一种保留为未指定（尽管保证是 `posix_close()` 的两种行为之一，而不是像标准早期版本那样完全未指定）。

请注意，标准要求 `close()` 和 `posix_close()` 必须在 [EINTR] 后保持 `fildes` 打开（在允许 [EINTR] 的情况下），并且必须关闭文件描述符，无论所有其他错误如何（除了 [EBADF]，其中 `fildes` 已经无效）。通常，可移植的应用程序只应在检查 [EINTR] 后重试 `close()`（并且在 `POSIX_CLOSE_RESTART` 被定义为零的实现上，此重试循环将是死代码），如果不尝试重试循环则有文件描述符泄漏的风险。还应注意，[EINTR] 只有在 `close()` 可以被中断时才可能；如果没有安装信号处理程序，那么 `close()` 不会被中断。相反，如果单线程应用程序可以保证在信号处理程序中没有文件

描述符被打开或关闭，那么不检查 [EINTR] 的重试循环将是无害的（因为重试将因 [EBADF] 而失败），但保证没有外部库引入线程的使用可能很困难。对于只会在 POSIX_CLOSE_RESTART 被定义为 0 的系统上使用的应用程序，还有额外的保证。这些观察应有助于确定应用程序是否需要将其 `close()` 调用审核为替换为 `posix_close()`。

还应注意，`posix_close()` 在 `flag` 为 0 时总是关闭 `fd` 的要求，即使在报告错误时也是如此，这与 `fclose()` 总是释放流的要求相似，即使在刷新数据时遇到错误也是如此。

以前总是关闭 `fd` 的实现可以通过在 `close()` 中将 [EINTR] 转换为 [EINPROGRESS] 来满足新要求；并且可以将 POSIX_CLOSE_RESTART 定义为 0 而不必添加重新启动语义。另一方面，以前在 [EINTR] 时保持 `fd` 打开的实现可以将其映射到带有 POSIX_CLOSE_RESTART 的 `posix_close()`，并且必须在 `flag` 为 0 时添加 `posix_close()` 的语义；一种可能性是调用原始的 `close()` 实现，检查失败，并在 [EINTR] 时，使用类似于 `dup2()` 的操作用另一个文件描述符替换不完整的关闭操作，该文件描述符可以通过对原始 `close()` 的另一次调用立即关闭，所有这些都在返回应用程序之前。无论哪种方式，`close()` 应始终映射到 `posix_close()` 的两种行为之一，并鼓励实现保持 `close()` 的行为不变，以免破坏依赖本标准早期版本未指定的行为的旧应用程序的实现特定期望。

标准开发人员考虑引入一个线程局部变量，`close()` 将设置该变量以指示它在返回 -1 时是否关闭了 `fd`。然而，这被拒绝，转而采用更简单的解决方案，即收紧 `close()` 以保证除了 [EINTR] 之外 `fd` 被关闭，并通过添加 `posix_close()` 来暴露是否期望 [EINTR] 的选择。此外，虽然名称 `posix_close()` 是本标准的新名称，但它让人联想到至少一个实现引入了名为 `close_nocancel()` 的替代系统调用，以允许应用程序选择是否需要重新启动语义。

另一个考虑是实现是否可能返回 [EAGAIN] 作为扩展，以及 `close()` 是否应被要求在这种情况下保持文件描述符打开，因为 [EAGAIN] 通常意味着操作应被重试。任何实现似乎都不太可能有正当理由返回 [EAGAIN] 或 [EWOULDBLOCK]，因此这个要求将意味着应用程序必须包含永远不会使用的错误情况的代码。因此 `close()` 现在被禁止返回 [EAGAIN] 和 [EWOULDBLOCK] 错误。

请注意，对套接字上的 `close()` 阻塞最多当前延迟间隔的要求不是以 `O_NONBLOCK` 设置为条件的。

标准开发人员拒绝了将 `closefrom()` 添加到标准的提议。因为标准允许实现使用继承的文件描述符作为子进程提供符合规范环境的手段，所以不可能标准化关闭高于某个值的任意文件描述符同时仍然保证符合规范环境的接口。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (另见)

- `dup()`
- `exec`
- `exit()`
- `fclose()`
- `fopen()`
- `fork()`
- `open()`
- `perror()`
- `unlink()`
- XBD `<unistd.h>`

CHANGE HISTORY (变更历史)

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

DESCRIPTION 为与 POSIX 实时扩展对齐而更新。

Issue 6

与基于 STREAMS 的文件或伪终端相关的 DESCRIPTION 被标记为 XSI STREAMS 选项组的一部分。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- [EIO] 错误条件作为可选错误添加。
- DESCRIPTION 更新为描述如果发生 I/O 错误并返回 [EIO] 错误条件，
`fildes` 文件描述符的状态为未指定。

关于套接字的文本被添加到 DESCRIPTION。

DESCRIPTION 为与 IEEE Std 1003.1j-2000 对齐而更新，指定共享内存对象和内存映射文件（而不是类型化内存对象）是最后一次关闭段落适用的内存对象类型。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/12，更正与伪终端管理方端相关的 XSH 阴影文本。变更的原因是，在这种情况下，伪终端和常规终端的行为应尽可能相似；变更实现了这一点并符合历史行为。

Issue 7

与 XSI STREAMS 选项相关的功能被标记为过时。

与异步输入和输出以及内存映射文件选项相关的功能被移至 Base。

应用 Austin Group Interpretation 1003.1-2001 #139，阐明对套接字上的 `close()` 阻塞最多当前延迟间隔的要求不是以 `O_NONBLOCK` 设置为条件的。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0059 [419], XSH/TC1-2008/0060 [149], 和 XSH/TC1-2008/0061 [149]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0069 [555]。

Issue 8

应用 Austin Group Defect 529，添加 `posix_close()` 函数并更改 `close()` 函数与 [EINTR] 相关的要求。

应用 Austin Group Defect 768，添加 OFD 拥有的文件锁。

应用 Austin Group Defect 1330，移除过时接口。

应用 Austin Group Defect 1466，更改用于伪终端设备的术语。

应用 Austin Group Defect 1525，阐明套接字直到与其关联的所有文件描述符都被关闭时才被销毁。

1.18. confstr — 获取可配置字符串变量

概要 (SYNOPSIS)

```
#include <unistd.h>

size_t confstr(int name, char *buf, size_t len);
```

描述 (DESCRIPTION)

`confstr()` 函数应返回配置定义的字符串值。其用途和目的与 `sysconf()` 类似，但用于返回字符串值而非数值。

`name` 参数表示要查询的系统变量。实现应支持以下在 `<unistd.h>` 中定义的 `name` 值，也可以支持其他值：

- `_CS_PATH`
- `_CS_POSIX_V8_ILP32_OFF32_CFLAGS`
- `_CS_POSIX_V8_ILP32_OFF32_LDFLAGS`
- `_CS_POSIX_V8_ILP32_OFF32_LIBS`
- `_CS_POSIX_V8_ILP32_OFFBIG_CFLAGS`
- `_CS_POSIX_V8_ILP32_OFFBIG_LDFLAGS`
- `_CS_POSIX_V8_ILP32_OFFBIG_LIBS`
- `_CS_POSIX_V8_LP64_OFF64_CFLAGS`
- `_CS_POSIX_V8_LP64_OFF64_LDFLAGS`
- `_CS_POSIX_V8_LP64_OFF64_LIBS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_CFLAGS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_LDFLAGS`
- `_CS_POSIX_V8_LPBIG_OFFBIG_LIBS`
- `_CS_POSIX_V8_THREADS_CFLAGS`
- `_CS_POSIX_V8_THREADS_LDFLAGS`
- `_CS_POSIX_V8_WIDTH_RESTRICTED_ENVS`
- `_CS_V8_ENV`

```
[OB] _CS_POSIX_V7_ILP32_OFF32_CFLAGS
[OB] _CS_POSIX_V7_ILP32_OFF32_LDFLAGS
[OB] _CS_POSIX_V7_ILP32_OFF32_LIBS
[OB] _CS_POSIX_V7_ILP32_OFFBIG_CFLAGS
[OB] _CS_POSIX_V7_ILP32_OFFBIG_LDFLAGS
[OB] _CS_POSIX_V7_ILP32_OFFBIG_LIBS
[OB] _CS_POSIX_V7_LP64_OFF64_CFLAGS
[OB] _CS_POSIX_V7_LP64_OFF64_LDFLAGS
[OB] _CS_POSIX_V7_LP64_OFF64_LIBS
[OB] _CS_POSIX_V7_LPBIG_OFFBIG_CFLAGS
[OB] _CS_POSIX_V7_LPBIG_OFFBIG_LDFLAGS
[OB] _CS_POSIX_V7_LPBIG_OFFBIG_LIBS
[OB] _CS_POSIX_V7_THREADS_CFLAGS
[OB] _CS_POSIX_V7_THREADS_LDFLAGS
[OB] _CS_POSIX_V7_WIDTH_RESTRICTED_ENVS
[OB] _CS_V7_ENV
```

如果 `len` 不为 0，且 `name` 有配置定义的值，`confstr()` 应将该值复制到由 `buf` 指向的长度为 `len` 字节的缓冲区中。如果要返回的字符串长度（包括终止空字符）超过 `len` 字节，那么 `confstr()` 应将字符串截断为 `len-1` 字节并将结果空字符终止。应用程序可以通过比较 `confstr()` 返回的值与 `len` 来检测字符串是否被截断。

如果 `len` 为 0 且 `buf` 为空指针，那么 `confstr()` 仍应返回如下定义的整数值，但不返回字符串。如果 `len` 为 0 但 `buf` 不是空指针，则结果未指定。

在调用以下代码后：

```
confstr(_CS_V8_ENV, buf, sizeof(buf))
```

存储在 `buf` 中的字符串应包含空格分隔的 `variable=值` 环境变量对列表，这些变量是实现作为指定符合性环境的一部分所需要的，如实现的符合性文档中所述。

如果实现支持 POSIX shell 选项，在调用以下代码后存储在 `buf` 中的字符串：

```
confstr(_CS_PATH, buf, sizeof(buf))
```

可以用作 `PATH` 环境变量的值来访问 POSIX.1-2024 的所有标准实用程序，这些实用程序的提供方式可通过 `exec` 函数族访问，前提是返回值小于或等于 `sizeof(buf)`。

返回值 (RETURN VALUE)

如果 `name` 有配置定义的值, `confstr()` 应返回保存整个配置定义值 (包括终止空字符) 所需的缓冲区大小。如果此返回值大于 `len`, 则返回在 `buf` 中的字符串被截断。

如果 `name` 无效, `confstr()` 应返回 0 并设置 `errno` 以指示错误。

如果 `name` 没有配置定义的值, `confstr()` 应返回 0 且保持 `errno` 不变。

错误 (ERRORS)

如果发生以下情况, `confstr()` 函数应失败:

- `[EINVAL]`
- `name` 参数的值无效。

以下各节为资料性内容。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

应用程序可以通过检查 `errno` 是否被修改来区分无效的 `name` 参数值和对应于没有配置定义值但可配置变量。这镜像了 `sysconf()` 的行为。

此函数的最初需求是提供一种方法来查找环境变量 `PATH` 的配置定义默认值。由于 `PATH` 可以被用户修改为包含可能包含替换 POSIX.1-2024 Shell 和实用程序卷中标准实用程序的实用程序的目录, 应用程序需要一种方法来确定系统提供的 `PATH` 环境变量值, 该值包含标准实用程序的正确搜索路径。

应用程序可以使用:

```
confstr(name, (char *)NULL, (size_t)0)
```

来找出字符串值需要多大的缓冲区; 使用 `malloc()` 分配缓冲区来保存字符串; 然后再次调用 `confstr()` 来获取字符串。或者, 它可以分配一个足够大

的固定静态缓冲区来保存大多数答案（可能是 512 或 1024 字节），但如果发现这个缓冲区太小，就使用 `malloc()` 分配更大的缓冲区。

基本原理 (RATIONALE)

应用程序开发人员通常可以通过读取由以下调用打开的流来确定任何配置变量：

```
popen("command -p getconf variable", "r");
```

`confstr()` 函数在 `name` 参数为 `_CS_PATH` 时返回一个字符串，该字符串可用作 `PATH` 环境变量设置，将引用 POSIX.1-2024 Shell 和实用程序卷中描述的标准 shell 和实用程序。

`confstr()` 函数将返回的字符串复制到应用程序提供的缓冲区中，而不是返回指向字符串的指针。这在某些实现（如具有轻量级线程的实现）中提供了更清洁的函数，并解决了应用程序何时必须复制返回字符串的问题。

未来方向 (FUTURE DIRECTIONS)

无。

另请参见 (SEE ALSO)

- `exec`
- `fpathconf()`
- `sysconf()`

XBD `<unistd.h>`

XCU `c17`

变更历史 (CHANGE HISTORY)

首次在 Issue 4 中发布。源自 ISO POSIX-2 标准。

Issue 5

在描述中添加了表示 `name` 允许值的表格。所有标记为 EX 的值在此版本中是新的。

Issue 6

应用了 The Open Group Corrigendum U033/7。返回缓冲区大小情况的返回值现在明确说明这包括终止空字符。

从与单一 UNIX 规范对齐得出以下对 POSIX 实现的新要求：

- 描述中更新了新参数，这些参数可用于确定每个不同支持编程环境的 C 编译器标志、链接器/加载器标志和库的配置字符串。这是为了支持数据大小中立性的更改。

做出了以下更改以与 IEEE P1003.1a 草案标准对齐：

- 描述中更新了文本，描述了如何使用 `_CS_PATH` 获取访问标准实用程序的 `PATH`。

与 `c89` 编程模型关联的宏被标记为 LEGACY，并引入了与 `c99` 关联的新等效宏。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #047，添加了 `_CS_V7_ENV` 变量。

应用了 Austin Group Interpretations 1003.1-2001 #166 以允许额外的编译器标志来启用线程。

支持的编程环境的 V6 变量被标记为过时的。

支持的编程环境的变量被更新为 V7。

LEGACY 变量和过时值被移除。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0070 [810] 和 XSH/TC2-2008/0071 [911]。

Issue 8

应用了 Austin Group Defect 1330，将 "V7" 更改为 "V8"，将 "V6" 更改为 "V7"。

1.19. `ctime` — 将时间值转换为日期和时间字符串

概要 (SYNOPSIS)

```
#include <time.h>

char *ctime(const time_t *clock);
```

描述 (DESCRIPTION)

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`ctime()` 函数应将 `clock` 指向的时间值（表示自纪元（Epoch）以来的秒数）转换为本地时间的字符串形式。它应等价于：

```
asctime(localtime(clock))
```

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解时间结构和一个 `char` 数组。执行任何返回指向这些对象类型之一的指针的函数可能会覆盖从之前对其中任何函数的调用所返回的值所指向的任何同类型对象中的信息。

`ctime()` 函数不需要是线程安全的；但是，`ctime()` 应避免与除自身、`asctime()`、`gmtime()` 和 `localtime()` 之外的所有函数发生数据竞争。

返回值 (RETURN VALUE)

`ctime()` 函数应返回以该分解时间作为参数调用 `asctime()` 所返回的指针。

错误 (ERRORS)

未定义错误。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

此函数仅用于与旧实现兼容。如果生成的字符串过长，它具有未定义行为，因此应不鼓励使用此函数。在未检测到输出字符串长度溢出的实现上，可能会以导致应用程序失败的方式溢出输出缓冲区，或者可能造成系统安全违规。此外，此函数不支持本地化的日期和时间格式。为避免这些问题，应用程序应使用 `strftime()` 从分解时间生成字符串。

分解时间结构的值可以通过调用 `gmtime()` 或 `localtime()` 来获得。

尝试对纪元之前的时间或 9999 年之后的时间使用 `ctime()` 会产生未定义结果。请参考 `asctime()`。

原理 (RATIONALE)

标准开发者决定将 `ctime()` 函数标记为过时，尽管它包含在 ISO C 标准中，原因是可能存在缓冲区溢出。ISO C 标准也提供了 `strftime()` 函数，可用于避免这些问题。

未来方向 (FUTURE DIRECTIONS)

此函数可能在将来的版本中被删除，但不会在从 ISO C 标准中删除之前删除。

参见 (SEE ALSO)

- `asctime()`
- `clock()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktimes()`

- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

- 以前在应用程序用法章节中的规范性文本被移至描述章节。
- 为与 POSIX 线程扩展对齐，包含了 `ctime_r()` 函数。
- 在描述章节中添加了说明 `ctime()` 函数不需要可重入的注释。

Issue 6

- 超出 ISO C 标准的扩展被标记。
- 在描述章节中，关于可重入性的注释被扩展以覆盖线程安全性。
- 应用程序用法章节被更新，包含关于线程安全函数及其避免可能使用静态数据区域的注释。

Issue 7

- 应用了 Austin Group 解释 1003.1-2001 #156。
- 应用了 SD5-XSH-ERN-25，更新了应用程序用法。
- 应用了 Austin Group 解释 1003.1-2001 #053，将这些函数标记为过时。
- `ctime_r()` 函数从线程安全函数选项移至基础。
- 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0066 [321,428]。
- 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0075 [664]。

Issue 8

- 应用了 Austin Group 缺陷 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。
 - 应用了 Austin Group 缺陷 1330，更改了未来方向章节。
 - 应用了 Austin Group 缺陷 1376，从源自 ISO C 标准的某些文本中移除了 CX 着色，并更新以匹配 ISO C 标准。
 - 应用了 Austin Group 缺陷 1410，移除了 `ctime_r()` 函数。
-

1.20. `ctime_r` — 将时间值转换为日期和时间字符串（已移除）

SYNOPSIS

```
#include <time.h>

char *ctime(const time_t *clock);
```

DESCRIPTION

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本标准遵循 ISO C 标准。

`ctime()` 函数应将 `clock` 指向的时间（表示自纪元以来以秒为单位的时间）转换为字符串形式的本地时间。它应等价于：

```
asctime(localtime(clock))
```

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解时间结构和一个 `char` 数组。执行任何返回指向这些对象类型之一指针的函数可能会覆盖任何先前调用其中任何函数所返回的值所指向的同类对象中的信息。

`ctime()` 函数不需要是线程安全的；但是，`ctime()` 应避免与除自身、`asctime()`、`gmtime()` 和 `localtime()` 之外的所有函数发生数据竞争。

RETURN VALUE

`ctime()` 函数应返回 `asctime()` 以该分解时间为参数时返回的指针。

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

此函数仅为了与较旧的实现兼容而包含。如果结果字符串过长，其行为未定义，因此应 discourage 使用此函数。在不检测输出字符串长度溢出的实现上，可能会以导致应用程序失败或可能造成系统安全违规的方式溢出输出缓冲区。此外，此函数不支持本地化的日期和时间格式。为避免这些问题，应用程序应使用 `strftime()` 从分解时间生成字符串。

分解时间结构的值可以通过调用 `gmtime()` 或 `localtime()` 获得。

尝试对纪元之前的时间或 9999 年之后的时间使用 `ctime()` 会产生未定义的结果。请参考 `asctime()`。

RATIONALE

标准开发者决定将 `ctime()` 函数标记为过时，尽管它在 ISO C 标准中，这是由于存在缓冲区溢出的可能性。ISO C 标准也提供了 `strftime()` 函数，可用于避免这些问题。

FUTURE DIRECTIONS

此函数可能在将来的版本中被移除，但不会在从 ISO C 标准中移除之前移除。

SEE ALSO

- `asctime()`
- `clock()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktimed()`

- `strftime()`
- `strptime()`
- `time()`
- XBD `<time.h>`

CHANGE HISTORY

首次发布于 Issue 1

源自 SVID 的 Issue 1。

Issue 5

- 之前位于 APPLICATION USAGE 部分的规范性文本移至 DESCRIPTION。
- 包含 `ctime_r()` 函数以与 POSIX 线程扩展对齐。
- 在 DESCRIPTION 中添加了指示 `ctime()` 函数不需要是可重入的说明。

Issue 6

- 标记了超出 ISO C 标准的扩展。
- 在 DESCRIPTION 中，关于可重入性的说明扩展为涵盖线程安全性。
- 更新了 APPLICATION USAGE 部分，以包含关于线程安全函数及其避免可能使用静态数据区域的说明。

Issue 7

- 应用了 Austin Group Interpretation 1003.1-2001 #156。
- 应用了 SD5-XSH-ERN-25，更新了 APPLICATION USAGE。
- 应用了 Austin Group Interpretation 1003.1-2001 #053，将这些函数标记为过时。
- `ctime_r()` 函数从线程安全函数选项移至基础部分。
- 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0066 [321,428]。
- 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0075 [664]。

Issue 8

- 应用了 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。
 - 应用了 Austin Group Defect 1330，更改了 FUTURE DIRECTIONS 部分。
 - 应用了 Austin Group Defect 1376，从源自 ISO C 标准的某些文本中移除 CX 着色并更新以匹配 ISO C 标准。
 - **应用了 Austin Group Defect 1410，移除了 `ctime_r()` 函数。**
-

1.21. difftime — 计算两个日历时间值之间的差值

概要 (SYNOPSIS)

```
#include <time.h>

double difftime(time_t time1, time_t time0);
```

描述 (DESCRIPTION)

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`difftime()` 函数应计算两个日历时间（由 `time()` 返回）之间的差值：
`time1 - time0`。

返回值 (RETURN VALUE)

`difftime()` 函数应返回以 `double` 类型表示的秒数差值。

错误 (ERRORS)

未定义任何错误。

以下章节为信息性内容。

示例 (EXAMPLES)

无。

应用用法 (APPLICATION USAGE)

无。

基本原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- `asctime()`
- `clock()`
- `ctime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktimes()`
- `strftime()`
- `strptime()`
- `time()`

XBD `<time.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 4。源自 ISO C 标准。

1.22. div

概要

```
#include <stdlib.h>

div_t div(int numer, int denom);
```

描述

`div()` 函数应计算分子 `numer` 除以分母 `denom` 的商和余数。如果除法不精确，结果的商是最接近代数商且绝对值较小的整数。如果结果无法表示，则行为是未定义的；否则，`quot*denom+rem` 应等于 `numer`。

返回值

`div()` 函数应返回一个 `div_t` 类型的结构体，其中包含商和余数。该结构体包含以下成员（顺序不限）：

```
int quot; /* 商 */
int rem; /* 余数 */
```

示例

未提供。

应用用法

未提供。

基本原理

未提供。

未来方向

未提供。

另请参见

未提供。

版权

本参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

1.23. exit

SYNOPSIS

```
#include <stdlib.h>

void exit(int status);
```

DESCRIPTION

`exit()` 函数应当导致正常进程终止的发生。不会调用通过 `at_quick_exit()` 函数注册的任何函数。如果一个进程多次调用 `exit()` 函数，或者在调用 `exit()` 函数的同时还调用 `quick_exit()` 函数，则其行为是未定义的。

`status` 的值可以是 0、`EXIT_SUCCESS`、`EXIT_FAILURE` 或任何其他值，不过只有最低的 8 位（即 `status` & 0377）可以从 `wait()` 和 `waitpid()` 获取；完整的值可以从 `waitid()` 和传递给 `SIGCHLD` 信号处理程序的 `siginfo_t` 中获取。

`exit()` 函数应当首先调用所有通过 `atexit()` 注册的函数，按其注册的相反顺序调用，但如果一个函数在注册时已有先前注册的函数已经被调用过，则该函数在这些之前注册的函数之后被调用。每个函数被调用的次数与它注册的次数相同。如果在调用任何此类函数的过程中，进行了 `longjmp()` 函数调用，该调用会终止对已注册函数的调用，则其行为是未定义的。

如果通过调用 `atexit()` 注册的函数未能返回，则剩余的已注册函数不会被调用，`exit()` 处理的其余部分也不会完成。

`exit()` 函数然后应当刷新所有带有未写入缓冲数据的打开流。对于每个作为其底层文件描述符的活动句柄，且文件尚未到达 EOF 且能够定位的流，底层打开文件描述的文件偏移量应当设置为该流的文件位置。对于每个打开的流，`exit()` 函数应当对与该流关联的文件描述符执行等效于 `close()` 的操作。最后，进程应当终止，具有与 `_进程终止的后果` 中描述的相同后果。

RETURN VALUE

`exit()` 函数不返回。

ERRORS

没有定义错误。

EXAMPLES

参见 APPLICATION USAGE。

APPLICATION USAGE

当 `exit()` 刷新带有未写入缓冲数据的流时，调用进程无法发现 `exit()` 是否成功将数据写入了底层文件描述符。因此，强烈建议应用程序在调用 `exit()` 或从 `main()` 的初始调用返回时，始终确保任何流中没有未写入的缓冲数据，并使用表示没有发生错误的 `status` 值。

例如，以下代码演示了一种确保 `stdout` 在使用状态 0 调用 `exit()` 之前已经成功刷新的方法。如果刷新失败，`stdout` 的底层文件描述符被关闭，这样 `exit()` 就不会尝试重复失败的写入操作。如果刷新成功，则执行使用 `ferror()` 的最终检查，以确保在之前的刷新操作中（当时未处理的）没有写入错误。

```
int status = 0;
if (fflush(stdout) != 0) {
    perror("appname: standard output");
    close(fileno(stdout));
    status = 1;
}
else if (ferror(stdout)) {
    fputs("appname: write error on standard output\n", stderr);
    status = 1;
}
exit(status);
```

另见 [_Exit\(\)](#)。

RATIONALE

参见 [_Exit\(\)](#)。

FUTURE DIRECTIONS

无。

SEE ALSO

- `_Exit()`
- `at_quick_exit()`
- `atexit()`
- `exec`
- `fflush()`
- `longjmp()`
- `quick_exit()`
- `tmpfile()`
- `wait()`
- `waitid()`
- `<stdlib.h>`

CHANGE HISTORY

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #031，将 `_Exit()` 和 `_exit()` 函数与 `exit()` 函数分离。

应用 Austin Group Interpretation 1003.1-2001 #085。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0100 [594]。

Issue 8

应用 Austin Group Defect 610，阐明 `exit()` 对打开流的影响。

应用 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准保持一致。

应用 Austin Group Defect 1490，更改 EXAMPLES 和 APPLICATION USAGE 部分。

应用 Austin Group Defect 1629，更改 APPLICATION USAGE 部分。

1.24. fclose

概要

```
#include <stdio.h>

int fclose(FILE *stream);
```

描述

`fclose()` 函数应该使 `stream` 指向的流被刷新，并且相关联的文件被关闭。流的任何未写入的缓冲数据应该写入文件；任何未读取的缓冲数据应该被丢弃。无论调用是否成功，流都应该与文件解除关联，任何由 `setbuf()` 或 `setvbuf()` 函数设置的缓冲区都应该与流解除关联。如果相关联的缓冲区是自动分配的，应该被释放。

如果文件尚未到达 EOF，并且文件是支持定位的，那么如果流是底层文件描述符的活动句柄，底层打开文件描述符的文件偏移量应该设置为流的文件位置。

如果流是可写的，并且有缓冲数据尚未写入文件，`fclose()` 函数应该标记更新底层文件的最后数据修改时间和最后文件状态改变时间戳。`fclose()` 函数应该执行等效于在与 `stream` 指向的流相关联的文件描述符上执行 `close()` 操作。

在调用 `fclose()` 之后，任何对 `stream` 的使用都会导致未定义行为。

返回值

成功完成后，`fclose()` 应该返回 0；否则，应该返回 EOF 并设置 `errno` 来指示错误。

错误

`fclose()` 函数在以下情况下可能失败：

- **[EAGAIN]**
- 为 `stream` 底层的文件描述符设置了 `O_NONBLOCK` 标志，并且线程将在写操作中被延迟。

- **[EBADF]**

- 流底层的文件描述符无效。

- **[EFBIG]**

- 尝试写入一个超过最大文件大小的文件。

- **[EFBIG]**

- 尝试写入一个超过进程文件大小限制的文件。同时应该为线程生成 SIGXFSZ 信号。

- **[EFBIG]**

- 文件是常规文件，并且尝试在或超过与相应流相关联的偏移量最大值的位置写入。

- **[EINTR]**

- `fclose()` 函数被信号中断。

- **[EIO]**

- 进程是后台进程组的成员，尝试向其控制终端写入，TOSTOP 被设置，调用线程未阻塞 SIGTTOU，进程未忽略 SIGTTOU，且进程的进程组是孤立的。此错误也可能在实现定义的条件下返回。

- **[ENOMEM]**

- 底层流由 `open_memstream()` 或 `open_wmemstream()` 创建，且可用内存不足。

- **[ENOSPC]**

- 包含文件的设备上没有剩余可用空间，或者 `fmemopen()` 函数使用的缓冲区中没有剩余空间。

- **[EPIPE]**

- 尝试写入管道或 FIFO，但没有进程为其打开读取。同时应该向线程发送 SIGPIPE 信号。

`fclose()` 函数在以下情况下可能失败：

- **[ENXIO]**

- 对不存在的设备发出请求，或者请求超出了设备的能力范围。

示例

无。

应用程序使用

由于在调用 `fclose()` 后，任何对 `stream` 的使用都会导致未定义行为，因此不应该在 `stdin`、`stdout` 或 `stderr` 上使用 `fclose()`，除非在进程终止前立即使用，以避免在其他依赖这些流的标准接口中触发未定义行为。如果应用程序注册了任何 `atexit()` 处理程序，这样的 `fclose()` 调用应该等到最后一个处理程序完成时才发生。一旦使用 `fclose()` 关闭了 `stdin`、`stdout` 或 `stderr`，就没有标准的方法重新打开这些流中的任何一个。

使用 `freopen()` 来改变 `stdin`、`stdout` 或 `stderr` 而不是关闭它们，可以避免文件在应用程序后期意外地作为特殊文件描述符 `STDIN_FILENO`、`STDOUT_FILENO` 或 `STDERR_FILENO` 之一被打开的危险。

原理

无。

未来方向

无。

参见

- 2.5 标准 I/O 流
- `atexit()`
- `close()`
- `fmemopen()`
- `fopen()`
- `freopen()`
- `getrlimit()`
- `open_memstream()`

- `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了大文件峰会扩展。

Issue 6

超出了 ISO C 标准的扩展被标记。

以下对 POSIX 实现的新要求来自于与单一 UNIX 规范的对齐：

- 添加了 [EFBIG] 错误作为大文件支持扩展的一部分。
- 添加了 [ENXIO] 可选错误条件。

描述部分被更新以说明无论调用是否成功，流和任何缓冲区都会被解除关联。这是为了与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/28，将错误部分中的 [EAGAIN] 错误从"进程将被延迟"更新为"线程将被延迟"。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #002，阐明了文件描述符和流的交互。

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 更新了 [ENOSPC] 错误条件并添加了 [ENOMEM] 错误。

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0113 [87], XSH/TC1-2008/0114 [79], 和 XSH/TC1-2008/0115 [14]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0104 [555]。

Issue 8

应用了 Austin Group Defect 308，阐明了 [EFBIG] 错误的处理。

应用了 Austin Group Defect 1669，从与进程文件大小限制相关的 [EFBIG] 错误部分移除了 XSI 着色。

1.25. fdatasync

SYNOPSIS (概要)

```
#include <unistd.h>

int fdatasync(int fildes);
```

DESCRIPTION (描述)

`fdatasync()` 函数应强制将与文件描述符 `fildes` 指示的文件相关联的所有当前排队 I/O 操作推进到同步 I/O 完成状态。

该功能应等价于定义了符号 `_POSIX_SYNCHRONIZED_IO` 的 `fsync()`，但区别在于所有 I/O 操作应按照同步 I/O 数据完整性完成的规定来完成。

RETURN VALUE (返回值)

如果成功，`fdatasync()` 函数应返回值 0；否则，函数应返回值 -1 并设置 `errno` 来指示错误。如果 `fdatasync()` 函数失败，不保证未完成的 I/O 操作已经完成。

ERRORS (错误)

`fdatasync()` 函数应在以下情况失败：

- **[EBADF]**
`fildes` 参数不是有效的文件描述符。
- **[EINVAL]**
此实现不支持对此文件的同步 I/O。

如果任何排队 I/O 操作失败，`fdatasync()` 应返回为 `read()` 和 `write()` 定义的错误条件。

以下部分为参考信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

注意，即使文件描述符不是以写入方式打开的，如果基础文件上有任何待处理的写入请求，那么在 `fdatasync()` 返回之前将完成这些 I/O 操作。

修改目录的应用程序（例如，通过在目录中创建文件）可以对该目录调用 `fdatasync()` 以确保目录条目被同步，但对于大多数应用程序来说，这应该是不必要的（参见 XBD 4.11 文件系统缓存）。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `aio_fsync()`
- `fcntl()`
- `fsync()`
- `open()`
- `read()`
- `write()`
- XBD `<unistd.h>`

CHANGE HISTORY (变更历史)

首次发布于第 5 版。为了与 POSIX 实时扩展对齐而包含。

Issue 6

[ENOSYS] 错误条件已被删除，因为如果实现不支持同步输入和输出选项，则无需提供存根。

`fdatasync()` 函数被标记为同步输入和输出选项的一部分。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0110 [501]。

Issue 8

应用了 Austin Group Defect 672，更改了应用程序使用部分。

参考文本结束。

1.26. fdopen

概要

```
[CX] #include <stdio.h>  
  
FILE *fdopen(int fildes, const char *mode);
```

描述

`fdopen()` 函数应将一个流与文件描述符关联。

`mode` 参数指向一个对 `fopen()` 有效的字符串。如果该字符串以以下字符之一开头，那么流应按照指定的方式与 `fildes` 关联。否则，行为是未定义的。

模式字符

'r'

- 如果 `mode` 包含 '+', 关联的流应打开用于更新 (读写)
- 否则，流应打开仅用于读取
- 如果 `fildes` 引用的打开文件描述设置了 O_APPEND，它应保持设置状态

'w'

- 如果 `mode` 包含 '+', 关联的流应打开用于更新 (读写)
- 否则，流应打开仅用于写入
- 文件不应被 `fdopen()` 调用截断
- 如果 `fildes` 引用的打开文件描述设置了 O_APPEND，它应保持设置状态

'a'

- 如果 `mode` 包含 '+', 关联的流应打开用于更新 (读写)
- 否则，流应打开仅用于写入
- 如果 `fildes` 引用的打开文件描述清除了 O_APPEND，是否通过 `fdopen()` 调用设置 O_APPEND 或保持清除状态是未指定的

附加模式选项

- `mode` 中存在 'x' 应没有效果
- 如果不存在 'e', `fildes` 的 FD_CLOEXEC 标志应保持不变；如果存在 'e', 应通过 `fdopen()` 调用设置
- 实现可能支持 `mode` 参数的附加值

要求

应用程序应确保通过 `mode` 参数表示的流模式被 `fildes` 引用的打开文件描述的文件访问模式所允许。

与新流关联的文件位置指示器设置为与文件描述符关联的文件偏移量指示的位置。

流的错误和文件结束指示器应被清除。`fdopen()` 函数可能导致基础文件的最后数据访问时间戳被标记为更新。

[SHM] 如果 `fildes` 引用共享内存对象，`fdopen()` 函数的结果是未指定的。

[TYM] 如果 `fildes` 引用类型化内存对象，`fdopen()` 函数的结果是未指定的。

`fdopen()` 函数应保留之前为对应于 `fildes` 的打开文件描述设置的最大偏移量。

返回值

成功完成后，`fdopen()` 应返回指向流的指针；否则，应返回空指针并设置 `errno` 以指示错误。

错误

`fdopen()` 函数应在以下情况下失败：

- [EMFILE] {STREAM_MAX} 流当前在调用进程中打开。

`fdopen()` 函数可能在以下情况下失败：

- [EBADF] `fildes` 参数不是有效的文件描述符。
- [EINVAL] `mode` 参数不是有效的模式。

- [EMFILE] {FOPEN_MAX} 流当前在调用进程中打开。
 - [ENOMEM] 空间不足以分配缓冲区。
-

以下部分是资料性的。

示例

无。

应用程序用法

文件描述符通过 `open()`、`dup()`、`creat()` 或 `pipe()` 等调用获得，这些调用打开文件但不返回流。

原理

文件描述符可能通过 `open()`、`creat()`、`pipe()`、`dup()`、`fcntl()` 或 `socket()` 获得；通过 `fork()`、`posix_spawn()` 或 `exec()` 继承；或者通过其他方式获得。

`fdopen()` 和 `fopen()` 的 `mode` 参数含义不同。对于 `fdopen()`，写入 ('w') 模式不能创建或截断文件，追加 ('a') 模式不能创建文件。为了与 `fopen()` 保持一致，允许在 `mode` 参数中包含 'b'；'b' 对结果流没有影响。关于指定追加 ('a') 模式是否在 O_APPEND 标志清除时设置该标志，实现有所不同，但鼓励这样做。由于 `fdopen()` 不创建文件，'x' 模式修饰符被静默忽略。'e' 模式修饰符对于 `fdopen()` 并非严格必要，因为当它不存在时 FD_CLOEXEC 不能被改变；但是，这里对其进行标准化，因为该修饰符对于避免使用 `freopen()` 的多线程应用程序中的数据竞争是必要的，并且一致性要求所有接受 `mode` 字符串的函数应该允许相同的字符串集合。

未来方向

无。

另见

- 2.5.1 文件描述符与标准 I/O 流的交互
- `fclose()`
- `fmemopen()`
- `fopen()`
- `open()`
- `open_memstream()`
- `posix_spawn()`
- `socket()`
- XBD

变更历史

在 Issue 1 中首次发布。源自 SVID 的 Issue 1。

Issue 5

- 更新了描述以与 POSIX 实时扩展对齐。
- 添加了大文件峰会扩展。

Issue 6

POSIX 实现的以下新要求源自与单一 UNIX 规范的对齐：

- 在描述中，`mode` 参数的使用和设置被更改为包含二进制流。
- 在描述中，添加了大文件支持的文本，以指示在打开文件描述中设置最大偏移量。
- 添加了错误部分中识别的所有错误。
- 在描述中，添加了 `fdopen()` 函数可能导致 `st_atime` 更新的文本。

为与 IEEE P1003.1a 草案标准对齐进行了以下更改：

- 添加了说明，确保模式与打开文件描述符兼容是应用程序的责任。

通过指定对于类型化内存对象 `fdopen()` 结果是未指定的，更新了描述以与 IEEE Std 1003.1j-2000 对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/30，对原理进行了更正。

Issue 7

- 应用了 SD5-XSH-ERN-149，添加了 {STREAM_MAX} [EMFILE] 错误条件。
- 进行了与细粒度时间戳支持相关的更改。
- 应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0121 [409]。

Issue 8

- 应用了 Austin Group 缺陷 411 和 1526，更改了 `mode` 参数的要求。
-

1.27. feclearexcept — 清除浮点异常

概要

```
#include <fenv.h>

int feclearexcept(int excepts);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`feclearexcept()` 函数应尝试清除由 `excepts` 表示的受支持的浮点异常。

返回值

如果参数为零或所有指定的异常都已成功清除，`feclearexcept()` 应返回零。否则，应返回非零值。

错误

未定义错误。

以下部分为参考信息。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参阅

- `fegetexceptflag()`
- `feraiseexcept()`
- `fetestexcept()`

XBD `<fenv.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准，技术勘误 1。

1.28. fegetenv, fesetenv — 获取和设置当前浮点环境

概要

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

描述

本参考页描述的功能与 ISO C 标准保持一致。如果此处描述的要求与 ISO C 标准之间存在冲突，则属无心之失。本卷 POSIX.1-2024 遵从 ISO C 标准。

`fegetenv()` 函数应尝试将当前浮点环境存储到 `envp` 指向的对象中。

`fesetenv()` 函数应尝试建立由 `envp` 指向的对象所表示的浮点环境。参数 `envp` 应指向通过调用 `fegetenv()` 或 `feholdexcept()` 设置的对象，或等于一个浮点环境宏。`fesetenv()` 函数不会引发浮点异常，只安装通过其参数表示的浮点状态标志的状态。

返回值

如果表示已成功存储，`fegetenv()` 应返回零。否则，应返回非零值。如果环境已成功建立，`fesetenv()` 应返回零。否则，应返回非零值。

错误

未定义任何错误。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参阅

- `feholdexcept()`
- `feupdateenv()`
- `<fenv.h>`

更改历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准，技术勘误 1。

1.29. fegetexceptflag, fesetexceptflag — 获取和设置浮点状态标志

SYNOPSIS

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非预期的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`fegetexceptflag()` 函数应尝试将由参数 `excepts` 指示的浮点状态标志的状态以实现定义的表示形式存储到由参数 `flagp` 指向的对象中。

`fesetexceptflag()` 函数应尝试将由参数 `excepts` 指示的浮点状态标志设置为存储在 `flagp` 指向的对象中的状态。 `flagp` 指向的值应已由先前调用 `fegetexceptflag()` 设置，其第二个参数至少表示了由参数 `excepts` 表示的那些浮点异常。此函数不会引发浮点异常，而只是设置标志的状态。

RETURN VALUE

如果表示形式成功存储，`fegetexceptflag()` 应返回零。否则，应返回非零值。

如果 `excepts` 参数为零或者所有指定的异常都成功设置，`fesetexceptflag()` 应返回零。否则，应返回非零值。

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `feclearexcept()`
- `feraiseexcept()`
- `fetestexcept()`
- XBD `<fenv.h>`

CHANGE HISTORY

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

ISO/IEC 9899:1999 标准的技术勘误 1 已被纳入。

1.30. fegetround, fesetround — 获取和设置当前舍入方向

概要

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本文遵循 ISO C 标准。

fegetround() 函数应获取当前的舍入方向。

fesetround() 函数应建立由其参数 **round** 表示的舍入方向。如果参数不等于舍入方向宏的值，则舍入方向不会改变。

返回值

fegetround() 函数应返回表示当前舍入方向的舍入方向宏的值，或者当没有相应的舍入方向宏或当前舍入方向无法确定时返回负值。

fesetround() 函数当且仅当所请求的舍入方向已建立时返回零值。

错误

未定义任何错误。

以下章节为补充信息。

示例

以下示例保存、设置和恢复舍入方向，如果设置舍入方向失败则报告错误并中止：

```
#include <fenv.h>
#include <assert.h>

void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;

    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

应用用法

无。

原理

无。

未来方向

无。

参见

XBD [`<fenv.h>`](#)

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

ISO/IEC 9899:1999 标准，技术勘误 1 已被纳入。

补充信息结束。

1.31. feholdexcept

概要

```
#include <fenv.h>

int feholdexcept(fenv_t *envp);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`feholdexcept()` 函数应将当前浮点环境保存在 `envp` 指向的对象中，清除浮点状态标志，然后为所有浮点异常安装非停止（浮点异常时继续运行）模式（如果可用）。

返回值

`feholdexcept()` 函数当且仅当成功安装非停止浮点异常处理时返回零。

错误

未定义错误。

以下部分为参考信息。

示例

无。

应用程序用法

无。

基本原理

`feholdexcept()` 函数在典型的 IEC 60559:1989 标准实现上应该是有效的，这些实现具有默认的非停止模式和至少一种其他模式用于陷阱处理或中止。如果实现只提供非停止模式，那么安装非停止模式就是简单的操作。

未来方向

无。

另请参阅

- `fegetenv()`
- `feupdateenv()`
- XBD `<fenv.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

1.32. feof

概要 (SYNOPSIS)

```
#include <stdio.h>

int feof(FILE *stream);
```

描述 (DESCRIPTION)

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`feof()` 函数应测试由 `stream` 指向的流的文件结束指示器。

[CX] 如果 `stream` 有效, `feof()` 函数不应改变 `errno` 的设置。

返回值 (RETURN VALUE)

当且仅当为 `stream` 设置了文件结束指示器时, `feof()` 函数应返回非零值。

错误 (ERRORS)

未定义错误。

以下章节为参考信息。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

无。

原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

另请参见 (SEE ALSO)

- `clearerr()`
- `ferror()`
- `fopen()`

XBD `<stdio.h>`

变更历史 (CHANGE HISTORY)

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008、技术勘误 1、XSH/TC1-2008/0124 [401]。

1.33. feraiseexcept — 引发浮点异常

概要

```
#include <fenv.h>

int feraiseexcept(int excepts);
```

描述

`feraiseexcept()` 函数应尝试引发由 `excepts` 参数表示的受支持的浮点异常。引发这些浮点异常的顺序是未指定的，但如果 `excepts` 参数表示 IEC 60559 中针对原子操作的有效并发浮点异常（即溢出和不精确，或下溢和不精确），则应先引发溢出或下溢，再引发不精确异常。`feraiseexcept()` 函数在引发溢出或下溢浮点异常时是否同时引发不精确浮点异常是由实现定义的。

返回值

如果参数为零，或者所有指定的异常都已成功引发，`feraiseexcept()` 应返回零。否则，应返回非零值。

错误

未定义任何错误。

示例

无。

应用用法

该效果旨在与算术操作引发的浮点异常类似。因此，由此函数引发的浮点异常的已启用陷阱会被触发。

基本原理

允许在引发溢出或下溢时同时引发不精确异常，因为在某些架构上，引发异常的唯一实用方法是执行具有该异常作为副作用的指令。该函数不限于仅接受原子操作的有效并发表达式，因此该函数可用于引发在多个操作中累积的异常。

未来方向

无。

另请参阅

- `feclearexcept()`
- `fegetexceptflag()`
- `fetestexcept()`
- `<fenv.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准，技术勘误 1。

Issue 7

已应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0111 [543]。

1.34. ferror — 测试流的错误指示器

SYNOPSIS (概要)

```
#include <stdio.h>

int ferror(FILE *stream);
```

DESCRIPTION (描述)

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 遵循 ISO C 标准。

`ferror()` 函数应测试由 `stream` 指向的流的错误指示器。

如果 `stream` 有效, `ferror()` 函数不应更改 `errno` 的设置。

RETURN VALUE (返回值)

当且仅当 `stream` 的错误指示器被设置时, `ferror()` 函数应返回非零值。

ERRORS (错误)

未定义错误。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `clearerr()`
- `feof()`
- `fopen()`

XBD `<stdio.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0125 [401]。

1.35. fegetenv, fesetenv — 获取和设置当前浮点环境

概要

```
#include <fenv.h>

int fegetenv(fenv_t *envp);
int fesetenv(const fenv_t *envp);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`fegetenv()` 函数应尝试将当前浮点环境存储在 `envp` 指向的对象中。

`fesetenv()` 函数应尝试建立由 `envp` 指向的对象所表示的浮点环境。参数 `envp` 应指向通过调用 `fegetenv()` 或 `feholdexcept()` 设置的对象，或者等于一个浮点环境宏。 `fesetenv()` 函数不会引发浮点异常，只是安装通过其参数表示的浮点状态标志的状态。

返回值

如果表示被成功存储，`fegetenv()` 应返回零。否则，它应返回非零值。如果环境被成功建立，`fesetenv()` 应返回零。否则，它应返回非零值。

错误

未定义错误。

示例

无。

应用用法

无。

原理

无。

未来方向

无。

另请参阅

- `feholdexcept()`
- `feupdateenv()`
- `<fenv.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准，技术勘误 1。

1.36. fegetexceptflag, fesetexceptflag — 获取和设置浮点状态标志

概要

```
#include <fenv.h>

int fegetexceptflag(fexcept_t *flagp, int excepts);
int fesetexceptflag(const fexcept_t *flagp, int excepts);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`fegetexceptflag()` 函数应尝试将由参数 `excepts` 指示的浮点状态标志的状态以实现定义的表示形式存储到由参数 `flagp` 指向的对象中。

`fesetexceptflag()` 函数应尝试将由参数 `excepts` 指示的浮点状态标志设置为存储在由参数 `flagp` 指向的对象中的状态。`flagp` 指向的值应由之前对 `fegetexceptflag()` 的调用设置，该调用的第二个参数至少表示了由参数 `excepts` 表示的那些浮点异常。此函数不会引发浮点异常，而只是设置标志的状态。

返回值

如果表示形式成功存储，`fegetexceptflag()` 应返回零。否则，应返回非零值。如果 `excepts` 参数为零或所有指定的异常都已成功设置，`fesetexceptflag()` 应返回零。否则，应返回非零值。

错误

未定义错误。

示例

无。

应用用法

无。

基本原理

无。

未来方向

无。

另请参见

- `feclearexcept()`
- `feraiseexcept()`
- `fetestexcept()`
- XBD `<fenv.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

ISO/IEC 9899:1999 标准的技术勘误 1 已被纳入。

1.37. fegetround, fesetround — 获取和设置当前舍入方向

概要

```
#include <fenv.h>

int fegetround(void);
int fesetround(int round);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

fegetround() 函数应获取当前的舍入方向。

fesetround() 函数应建立由其参数 **round** 表示的舍入方向。如果参数不等于舍入方向宏的值，则舍入方向不会改变。

返回值

fegetround() 函数应返回表示当前舍入方向的舍入方向宏的值，如果不存在此类舍入方向宏或当前舍入方向无法确定，则返回负值。

fesetround() 函数当且仅当建立了请求的舍入方向时才返回零值。

错误

未定义错误。

以下章节为信息性内容。

示例

以下示例保存、设置和恢复舍入方向，如果设置舍入方向失败则报告错误并中止：

```
#include <fenv.h>
#include <assert.h>
void f(int round_dir)
{
    #pragma STDC FENV_ACCESS ON
    int save_round;
    int setround_ok;
    save_round = fegetround();
    setround_ok = fesetround(round_dir);
    assert(setround_ok == 0);
    /* ... */
    fesetround(save_round);
    /* ... */
}
```

应用用法

无。

基本原理

无。

未来方向

无。

另请参阅

XBD [`<fenv.h>`](#)

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准，技术勘误 1。

信息性文本结束。

1.38. fetestexcept — 测试浮点异常标志

概要

```
#include <fenv.h>

int fetestexcept(int excepts);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

fetestexcept() 函数应确定指定浮点异常标志子集中当前已设置的标志。
excepts 参数指定要查询的浮点状态标志。

返回值

fetestexcept() 函数应返回与 **excepts** 中包含的当前已设置浮点异常相对应的浮点异常宏的按位或运算值。

错误

未定义错误。

以下部分为参考信息。

示例

以下示例在设置了无效异常时调用函数 **f()**，然后在设置了溢出异常时调用函数 **g()**：

```
#include <fenv.h>
/* ... */
{
```

```
#pragma STDC FENV_ACCESS ON
int set_excepts;
feclearexcept(FE_INVALID | FE_OVERFLOW);
// 可能引发异常
set_excepts = fetestexcept(FE_INVALID | FE_OVERFLOW);
if (set_excepts & FE_INVALID) f();
if (set_excepts & FE_OVERFLOW) g();
/* ... */
}
```

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参阅

- [feclearexcept\(\)](#)
- [fegetexceptflag\(\)](#)
- [feraiseexcept\(\)](#)

XBD [<fenv.h>](#)

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

1.39. feupdateenv — 更新浮点环境

概要

```
#include <fenv.h>
int feupdateenv(const fenv_t *envp);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`feupdateenv()` 函数应尝试将当前触发的浮点异常保存在其自动存储中，尝试安装由 `envp` 指向的对象所表示的浮点环境，然后尝试触发保存的浮点异常。参数 `envp` 应指向通过调用 `feholdexcept()` 或 `fegetenv()` 设置的对象，或等于一个浮点环境宏。

返回值

`feupdateenv()` 函数应返回零值当且仅当所有必需的操作都成功执行。

错误

未定义错误。

以下章节为参考信息。

示例

以下示例展示了隐藏虚假下溢浮点异常的示例代码：

```
#include <fenv.h>
double f(double x)
{
    #pragma STDC FENV_ACCESS ON
    double result;
    fenv_t save_env;
    feholdexcept(&save_env);
```

```
// 计算结果
if /* 测试虚假下溢 */
feclearexcept(FE_UNDERFLOW);
feupdateenv(&save_env);
return result;
}
```

应用用法

无。

基本原理

无。

未来方向

无。

参见

- [fegetenv\(\)](#)
- [feholdexcept\(\)](#)
- XBD [`<fenv.h>`](#)

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

已纳入 ISO/IEC 9899:1999 标准技术勘误 1。

1.40. `fflush` — 刷新流

概要

```
#include <stdio.h>

int fflush(FILE *stream);
```

描述

本参考页中描述的功能与 ISO C 标准一致。如果此处描述的要求与 ISO C 标准之间存在冲突，则非有意为之。本 POSIX.1-2024 卷遵循 ISO C 标准。

如果 `stream` 指向输出流或更新流且最近操作不是输入，`fflush()` 应使该流的任何未写入数据被写入文件，底层文件的最后数据修改和最后文件状态更改时间戳应被标记为更新。

对于打开用于读取且具有底层文件描述的流，如果文件尚未到达 EOF，并且该文件是可寻址的，底层打开文件描述的文件偏移量应设置为流的文件位置，由 `ungetc()` 或 `ungetwc()` 推回流中且尚未从流中读取的任何字符应被丢弃（不再进一步更改文件偏移量）。

如果 `stream` 是空指针，`fflush()` 应对所有行为如上所述定义的流执行此刷新操作。

返回值

成功完成后，`fflush()` 应返回 0；否则，应设置流的错误指示器，返回 EOF，并设置 `errno` 以指示错误。

错误

`fflush()` 函数可能失败，如果：

- **[EAGAIN]**
- 为 `stream` 底层的文件描述符设置了 `O_NONBLOCK` 标志，且线程将在写操作中延迟。
- **[EBADF]**

- `stream` 底层的文件描述符无效。
- **[EFBIG]**
 - 尝试写入超过最大文件大小的文件。
- **[EFBIG]**
 - 尝试写入超过进程文件大小限制的文件。还应为该线程生成 SIGXFSZ 信号。
- **[EFBIG]**
 - 文件是常规文件，且尝试写入等于或超过与相应流关联的偏移量最大值。
- **[EINTR]**
 - `fflush()` 函数被信号中断。
- **[EIO]**
 - 进程是后台进程组的成员，尝试写入其控制终端，TOSTOP 已设置，调用线程未阻塞 SIGTTOU，进程未忽略 SIGTTOU，且进程的进程组是孤立的。此错误也可能在实现定义的条件下返回。
- **[ENOMEM]**
 - 底层流由 `open_memstream()` 或 `open_wmemstream()` 创建，且可用内存不足。
- **[ENOSPC]**
 - 包含文件的设备上或 `fmemopen()` 函数使用的缓冲区中没有剩余可用空间。
- **[EPIPE]**
 - 尝试写入没有任何进程打开用于读取的管道或 FIFO。还应向该线程发送 SIGPIPE 信号。

`fflush()` 函数可能失败，如果：

- **[ENXIO]**
 - 向不存在的设备发出请求，或请求超出设备的能力范围。

示例

向标准输出发送提示

以下示例使用 `printf()` 调用打印一系列提示，用户必须从标准输入输入信息。`fflush()` 调用强制输出到标准输出。使用 `fflush()` 函数是因为标准输出通常是缓冲的，提示可能不会立即打印到输出或终端上。`getline()` 函数调用从标准输入读取字符串并将结果放在变量中，供程序稍后使用。

```
char *user;
char *oldpasswd;
char *newpasswd;
ssize_t llen;
size_t blen;
struct termios term;
tcflag_t saveflag;

printf("User name: ");
fflush(stdout);
blen = 0;
llen = getline(&user, &blen, stdin);
user[llen-1] = 0;
tcgetattr(fileno(stdin), &term);
saveflag = term.c_lflag;
term.c_lflag &= ~ECHO;
tcsetattr(fileno(stdin), TCSANOW, &term);
printf("Old password: ");
fflush(stdout);
blen = 0;
llen = getline(&oldpasswd, &blen, stdin);
oldpasswd[llen-1] = 0;

printf("\nNew password: ");
fflush(stdout);
blen = 0;
llen = getline(&newpasswd, &blen, stdin);
newpasswd[llen-1] = 0;
term.c_lflag = saveflag;
tcsetattr(fileno(stdin), TCSANOW, &term);
free(user);
free(oldpasswd);
free(newpasswd);
```

应用程序使用

无。

原理

系统缓冲的数据可能使确定当前文件描述符位置的有效性变得不切实际。因此，POSIX.1-2024 不强制要求在打开用于 `read()` 的流上 `fflush()` 后重新定位文件描述符。

未来方向

无。

另请参阅

- 2.5 标准 I/O 流
- `fmemopen()`
- `getrlimit()`
- `open_memstream()`
-

更改历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了大文件峰会扩展。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源自与单一 UNIX 规范的对齐：

- 添加了 [EFBIG] 错误作为大文件支持扩展的一部分。

- 添加了 [ENXIO] 可选错误条件。

更新了 RETURN VALUE 部分，以注意应为流设置错误指示器。这是为了与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/31，将 ERRORS 部分中的 [EAGAIN] 错误从"进程将被延迟"更新为"线程将被延迟"。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #002，阐明了文件描述符和流的交互。

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 更新了 [ENOSPC] 错误条件并添加了 [ENOMEM] 错误。

修订了 EXAMPLES 部分。

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0126 [87], XSH/TC1-2008/0127 [79], 和 XSH/TC1-2008/0128 [14]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0112 [816] 和 XSH/TC2-2008/0113 [626]。

Issue 8

应用了 Austin Group Defect 308，阐明了 [EFBIG] 错误的处理。

应用了 Austin Group Defect 1669，从 [EFBIG] 错误中删除了与进程文件大小限制相关的 XSI 着色。

1.41. fgetc — 从流中获取一个字节

SYNOPSIS

```
#include <stdio.h>

int fgetc(FILE *stream);
```

DESCRIPTION

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 遵从 ISO C 标准。

如果指向 `stream` 的输入流的文件结束指示符未设置且存在下一个字节，`fgetc()` 函数应从指向 `stream` 的输入流中获取下一个字节作为转换为 `int` 的 `unsigned char`，并推进该流的相关文件位置指示符（如果已定义）。由于 `fgetc()` 操作字节，读取由多个字节组成的字符（或“多字节字符”）可能需要多次调用 `fgetc()`。

`fgetc()` 函数可以标记与 `stream` 关联的文件的最后数据访问时间戳以进行更新。最后一次数据访问时间戳应在第一次成功执行 `fgetc()`、`fgets()`、`fread()`、`fscanf()`、`getc()`、`getchar()`、`getdelim()`、`getline()` 或 `scanf()` 时标记为更新，这些函数使用 `stream` 返回的数据不是由先前的 `ungetc()` 调用提供的。

RETURN VALUE

成功完成时，`fgetc()` 应返回从指向 `stream` 的输入流中获取的下一个字节。如果流的文件结束指示符已设置，或者流处于文件结束状态，流的文件结束指示符应被设置，且 `fgetc()` 应返回 EOF。如果发生错误，流的错误指示符应被设置，`fgetc()` 应返回 EOF，并应设置 `errno` 以指示错误。

ERRORS

如果需要读取数据，`fgetc()` 函数将失败：

- **[EAGAIN]** 为 `stream` 底层的文件描述符设置了 `O_NONBLOCK` 标志, 线程将在 `fgetc()` 操作中被延迟。
- **[EBADF]** `stream` 底层的文件描述符不是用于读取的有效文件描述符。
- **[EINTR]** 读取操作因接收到信号而终止, 且没有数据传输。
- **[EIO]** 发生物理 I/O 错误, 或者进程处于后台进程组中尝试从其控制终端读取, 并且调用线程正在阻塞 `SIGTTIN`, 或者进程正在忽略 `SIGTTIN`, 或者进程的进程组是孤立的。此错误也可能因实现定义的原因而产生。
- **[EOVERFLOW]** 文件是常规文件, 且尝试在或超过相应流关联的偏移量最大值处读取。

在以下情况下 `fgetc()` 函数可能失败:

- **[ENOMEM]** 可用存储空间不足。
- **[ENXIO]** 对不存在的设备发出请求, 或者请求超出了设备的能力。

EXAMPLES

无。

APPLICATION USAGE

如果将 `fgetc()` 返回的整数值存储到 `char` 类型的变量中, 然后与整数常量 `EOF` 进行比较, 比较可能永远不会成功, 因为 `char` 类型的变量在扩展为整数时的符号扩展是实现定义的。

必须使用 `ferror()` 或 `feof()` 函数来区分错误条件和文件结束条件。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准 I/O 流

- `feof()`

- `ferror()`

- `fgets()`

- `fread()`

- `fscanf()`

- `getchar()`

- `getc()`

- `ungetc()`

- XBD `<stdio.h>`

CHANGE HISTORY

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

添加了大文件峰会扩展。

Issue 6

标记了超出 ISO C 标准的扩展。

POSIX 实现的以下新要求来源于与单一 UNIX 规范的对齐：

- 添加了 [EIO] 和 [EOVERFLOW] 强制错误条件。
- 添加了 [ENOMEM] 和 [ENXIO] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐进行了以下更改：

- 更新了 DESCRIPTION 以阐明输入流的文件结束指示符未设置时的行为。
- 更新了 RETURN VALUE 部分，指出应设置流的错误指示符。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/32，将 ERRORS 部分中的 [EAGAIN] 错误从"进程将被延迟"更新为"线程将被延迟"。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #051，更新了标记最后数据访问时间戳以进行更新的函数列表。

应用了 POSIX.1-2008，技术勘误表 1，XSH/TC1-2008/0129 [79] 和 XSH/TC1-2008/0130 [14]。

Issue 8

应用了 Austin Group Defect 1330，移除了过时的接口。

应用了 Austin Group Defect 1624，更改了 RETURN VALUE 部分。

1.42. fgets

概要

```
#include <stdio.h>

char *fgets(char *restrict s, int n, FILE *restrict stream);
```

描述

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`fgets()` 函数应从 `stream` 中读取字节到由 `s` 指向的数组中，直到读取了 `n` -1 个字节，或读取了 `<newline>` (换行符) 并传输到 `s`，或遇到文件结束条件。应在读取到数组中的最后一个字节之后立即写入一个空字节。如果在读取任何字节之前遇到文件结束条件，则由 `s` 指向的数组内容不应更改。

`fgets()` 函数可能会标记与 `stream` 关联的文件的最后数据访问时间戳以供更新。第一次成功执行使用 `stream` 并返回非先前调用 `ungetc()` 所提供数据的 `fgetc()`、`fgets()`、`fread()`、`fscanf()`、`getc()`、`getchar()`、`getdelim()`、`getline()` 或 `scanf()` 时，应标记最后数据访问时间戳以供更新。

返回值

成功完成后，`fgets()` 应返回 `s`。如果流位于文件结尾，应设置流的文件结束指示器，`fgets()` 应返回空指针。如果发生错误，应设置流的错误指示器，`fgets()` 应返回空指针，并应设置 `errno` 以指示错误。

错误

参考 `fgetc()`。

示例

读取输入

以下示例使用 `fgets()` 读取输入行。它假定正在读取的文件是文本文件，并且该文本文件中的行长度不超过 16384 字节（或在运行的实现上，如果 `{LINE_MAX}` 小于 16384，则为 `{LINE_MAX}`）。（注意，如果 `sysconf(_SC_LINE_MAX)` 返回 -1 且未设置 `errno`，则标准工具没有行长度限制。此示例假设 `sysconf(_SC_LINE_MAX)` 不会失败。）

```
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

#define MYLIMIT 16384

char *line;
int line_max;

if (LINE_MAX >= MYLIMIT) {
    /* 使用 MYLIMIT 的最大行大小。如果 LINE_MAX
       大于我们的限制，sysconf() 无法报告
       更小的限制。 */
    line_max = MYLIMIT;
} else {
    long limit = sysconf(_SC_LINE_MAX);
    line_max = (limit < 0 || limit > MYLIMIT) ? MYLIMIT : (int)
}

/* line_max + 1 为 fgets() 添加的空字节留出空间。 */
line = malloc(line_max + 1);
if (line == NULL) {
    /* 空间不足 */
    ...
    return error;
}

while (fgets(line, line_max + 1, fp) != NULL) {
    /* 验证是否已读取完整行... */
    /* 如果没有，报告错误或准备将下一次循环
       视为读取当前行的继续。 */
    ...
    /* 处理行... */
    ...
}
```

```
free(line);
```

```
...
```

应用用法

无。

原理

无。

未来方向

无。

另请参见

- 2.5 标准 I/O 流

- [fgetc\(\)](#)
- [fopen\(\)](#)
- [fread\(\)](#)
- [fscanf\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [getdelim\(\)](#)
- [ungetc\(\)](#)

• XBD [<stdio.h>](#)

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

`fgets()` 的原型已更改以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #051，更新了标记最后数据访问时间戳以供更新的函数列表。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0134 [182] 和 XSH/TC1-2008/0135 [14]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0114 [468]。

Issue 8

应用了 Austin Group Defect 1330，移除了过时的接口。

应用了 Austin Group Defect 1624，更改了 RETURN VALUE 部分。

1.43. fileno — 将流指针映射为文件描述符

概要

```
#include <stdio.h>

int fileno(FILE *stream);
```

描述

`fileno()` 函数应返回与 `stream` 参数指向的流相关联的整数文件描述符。

返回值

成功完成后, `fileno()` 应返回与 `stream` 相关联的文件描述符的整数值。
否则, 应返回值 -1 并设置 `errno` 以指示错误。

错误

在以下情况下, `fileno()` 函数应失败:

- **[EBADF]**
流与文件无关联。

在以下情况下, `fileno()` 函数可能失败:

- **[EBADF]**
`stream` 底层的文件描述符不是有效的文件描述符。

示例

无。

应用程序用法

无。

原理说明

如果没有规范说明哪些文件描述符与这些流相关联，应用程序就无法为通过 `fork()` 和 `exec` 启动的另一个应用程序设置流。特别是，将无法编写可移植版本的 `sh` 命令解释器（尽管可能有其他约束会阻止这种可移植性）。

未来方向

无。

参见

- 2.5.1 文件描述符与标准 I/O 流的交互
- `dirfd()`
- `fdopen()`
- `fopen()`
- `stdin`
- XBD `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

对 POSIX 实现的以下新要求源于与单一 UNIX 规范的对齐：

- 添加了 [EBADF] 可选错误条件。

Issue 7

应用了 SD5-XBD-ERN-99，更改了 [EBADF] 错误的定义。

应用了 POSIX.1-2008 Technical Corrigendum 2，XSH/TC2-2008/0115 [589]。

1.44. flockfile, ftrylockfile, funlockfile — stdio 锁定函数

SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

DESCRIPTION

这些函数为与标准 I/O 流（参见 2.5 标准 I/O 流）相关联的锁提供明确的应用程序级锁定。线程可以使用这些函数来划分作为一个单元执行的 I/O 语句序列。

`flockfile()` 函数应为线程获取 (`FILE *`) 对象的所有权。

`ftrylockfile()` 函数应在对象可用时为线程获取 (`FILE *`) 对象的所有权；
`ftrylockfile()` 是 `flockfile()` 的非阻塞版本。

`funlockfile()` 函数应放弃授予线程的所有权。如果当前所有者以外的线程调用 `funlockfile()` 函数，行为是未定义的。

这些函数的行为应表现为每个 (`FILE *`) 对象都关联有一个锁计数器。当 (`FILE *`) 对象创建时，该计数器被隐式初始化为零。当计数器为零时，(`FILE *`) 对象处于未锁定状态。当计数器为正数时，单个线程拥有 (`FILE *`) 对象。当调用 `flockfile()` 函数时，如果计数器为零，或者计数器为正数且调用者拥有 (`FILE *`) 对象，则计数器应递增。否则，调用线程应被挂起，等待计数器返回零。每次调用 `funlockfile()` 都应递减计数器。这使得匹配的 `flockfile()`（或成功的 `ftrylockfile()`）调用和 `funlockfile()` 调用可以嵌套。

RETURN VALUE

`flockfile()` 和 `funlockfile()` 没有返回值。

`ftrylockfile()` 函数成功时应返回零，非零值表示无法获取锁。

ERRORS

未定义错误。

APPLICATION USAGE

使用这些函数的应用程序可能会受到优先级反转的影响，如 XBD 3.275 优先级反转中所述。

对 `exit()` 的调用可能会阻塞，直到锁定的流被解锁，因为拥有 (`FILE*`) 对象所有权的线程会阻止其他线程引用该 (`FILE*`) 对象的所有函数调用（名称以 `_unlocked` 结尾的函数除外），包括对 `exit()` 的调用。

注意：FILE 锁不是文件锁（参见 XBD 3.143 文件锁）。

RATIONALE

`flockfile()` 和 `funlockfile()` 函数为每个 FILE 提供一个正交互斥锁。`ftrylockfile()` 函数提供获取 FILE 锁的非阻塞尝试，类似于 `pthread_mutex_trylock()`。

这些锁的行为表现为与 stdio 内部用于线程安全的锁相同。这既提供了这些函数的线程安全，又不需要第二级内部锁定，并允许 stdio 中的函数用其他 stdio 函数来实现。

应用程序开发者和实现者应该注意，FILE 对象存在潜在的死锁问题。例如，stdio 的行缓冲刷新语义（通过 `{_IOLBF}` 请求）要求某些输入操作有时导致实现定义的行缓冲输出流的缓冲内容被刷新。如果两个线程各自持有对方 FILE 的锁，就会发生死锁。通过以一致的顺序获取 FILE 锁可以避免这种类型的死锁。特别是，如果线程既要获取输入流又要获取输出流的锁，通常可以通过在获取输出流锁之前获取输入流锁来避免行缓冲输出流死锁。

总之，与其他线程共享 stdio 流的线程可以使用 `flockfile()` 和 `funlockfile()` 来保持单个线程执行的 I/O 序列捆绑在一起。需要使用 `flockfile()` 和 `funlockfile()` 的唯一情况是提供保护 `*_unlocked` 函数/宏使用的作用域。这会将成本/性能权衡移动到最佳点。

FUTURE DIRECTIONS

无。

SEE ALSO

- `exit()`
- `getc_unlocked()`
- XBD 3.275 优先级反转
- `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

Issue 7

`flockfile()`、`ftrylockfile()` 和 `funlockfile()` 函数从线程安全函数选项移动到基础标准。

应用 POSIX.1-2008 技术勘误表 1，XSH/TC1-2008/0140 [118]。

应用 POSIX.1-2008 技术勘误表 2，XSH/TC2-2008/0116 [611]。

Issue 8

应用 Austin Group 缺陷 1118，明确说明 FILE 锁不是文件锁。

应用 Austin Group 缺陷 1302，用对 2.5 标准 I/O 流的引用替换部分文本。

1.45. fopen — 打开流

概要

```
#include <stdio.h>

FILE *fopen(const char *restrict pathname, const char *restrict
```

描述

除了"独占访问"要求（见下文）外，此参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何其他冲突都是无意的。本 POSIX.1-2024 卷在涉及"独占访问"方面之外，所有 `fopen()` 功能均遵循 ISO C 标准。

`fopen()` 函数应打开路径名为 `pathname` 指向的字符串所指定的文件，并将一个流与之关联。

`mode` 参数指向一个字符串。如果字符串中的任何字符出现多次，则行为未指定。如果字符串以以下字符之一开头，则文件应以指示的模式打开。否则，行为未定义。

- `'r'`: 打开文件用于读取。
- `'w'`: 截断为零长度或创建文件用于写入。
- `'a'`: 追加；在文件末尾打开或创建文件用于写入。

字符串的其余部分可以包含以下任意字符，以任意顺序，并进一步影响文件的打开方式：

- `'b'`: 此字符应无效，但允许符合 ISO C 标准。
- `'e'`: 底层文件描述符应原子地设置 `FD_CLOEXEC` 标志。
- `'x'`: 如果 `mode` 的第一个字符是 `'w'` 或 `'a'`，则如果文件已存在或无法创建，函数应失败；如果文件不存在且可以创建，则应在底层文件系统支持的情况下，以实现定义的独占（也称为非共享）访问形式创建文件，前提是生成的文件权限与不使用 `'x'` 修饰符时的权限相同。如果 `mode` 的第一个字符是 `'r'`，则效果是实现定义的。

注意：

ISO C 标准要求独占访问"在底层文件系统支持独占访问的范围内"，但没有定义这意味着什么。按字面理解——系统必须在文件系统级别尽其所能来排除他人

的访问——这将要求 POSIX.1 系统以防止其他用户和组访问的方式设置文件权限。因此，本 POSIX.1-2024 卷在“独占访问”要求方面不遵循 ISO C 标准。

- '+': 文件应打开用于更新（读取和写入），而不仅仅是读取或写入。

以读取模式打开文件（`mode` 参数中的第一个字符为 'r'）应失败，如果文件不存在或无法读取。

以追加模式打开文件（`mode` 参数中的第一个字符为 'a'）应导致所有后续对文件的写入都被强制到当前文件末尾，无论中间是否有 `fseek()` 调用。

当以更新模式打开文件（`mode` 参数中包含 '+'）时，可以在关联的流上执行输入和输出操作。但是，应用程序应确保输出之后不直接跟随输入，除非有中间的 `fflush()` 调用或文件定位函数（`fseek()`、`fsetpos()` 或 `rewind()`），并且输入之后不直接跟随输出，除非有中间的文件定位函数调用，除非输入操作遇到文件末尾。

打开时，当且仅当可以确定流不引用交互设备时，流才是完全缓冲的。流的错误和文件结束指示器应被清除。

如果 `mode` 中的第一个字符是 'w' 或 'a' 且文件先前不存在，则在成功完成时，`open()` 应标记更新文件的最后数据访问、最后数据修改和最后文件状态更改时间戳，以及父目录的最后文件状态更改和最后数据修改时间戳。

如果 `mode` 中的第一个字符是 'w' 或 'a' 且文件先前不存在，`open()` 函数应创建文件，就像它调用了 `open()` 函数，其中 `path` 参数从 `pathname` 解释，`oflag` 参数值如下指定，第三个参数值为 `S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH`。

如果 `mode` 中的第一个字符是 'w' 且文件先前存在，则在成功完成时，`open()` 应标记更新文件的最后数据修改和最后文件状态更改时间戳。

成功调用 `open()` 函数后，流的定向应被清除，编码规则应被清除，关联的 `mbstate_t` 对象应被设置为描述初始转换状态。

与打开的流关联的文件描述符应分配并打开，就像通过使用以下标志调用 `open()` 一样：

<code>fopen()</code> 模式第一个字符	<code>fopen()</code> 模式包含 '+'	初始 <code>open()</code> 标志
'r'	否	<code>O_RDONLY</code>
'w'	否	<code>O_WRONLY</code>
'a'	否	<code>O_WRONLY</code>

<code>fopen()</code> 模式第一个字符	<code>fopen()</code> 模式包含 '+'	初始 <code>open()</code> 标志
'r'	是	O_RDWR
'w'	是	O_RDWR
'a'	是	O_RDWR

当且仅当指定了 'e' 模式字符串字符时，O_CLOEXEC 标志应与上述表中指定的初始 `open()` 标志进行 OR 运算。

当且仅当 'x' 模式字符串字符与 'w' 或 'a' 一起指定时，O_EXCL 标志应与上述表中指定的初始 `open()` 标志进行 OR 运算。

如果 mode 包含 'x' 且底层文件系统支持通过使用实现特定的 `open()` 标志启用独占访问（见上文），则行为应如同也包含了这些标志。

当使用本标准指定的模式字符串时，实现应表现为没有其他标志传递给 `open()`。

返回值

成功完成时，`fopen()` 应返回指向控制流的对象的指针。否则，应返回空指针，并设置 `errno` 以指示错误。

错误

`fopen()` 函数应在以下情况下失败：

- **[EACCES]**: 路径前缀的某个组件的搜索权限被拒绝，或文件存在且 mode 指定的权限被拒绝，或文件不存在且要创建文件的父目录的写权限被拒绝。
- **[EEXIST]**: mode 参数以 w 或 a 开头且包含 x，但文件已存在。
- **[EILSEQ]**: mode 参数以 w 或 a 开头，文件先前不存在，且最后一个路径名组件不是可移植文件名且无法在目标目录中创建。
- **[EINTR]**: 在 `fopen()` 期间捕获了信号。
- **[EISDIR]**: 命名文件是目录且 mode 要求写访问。
- **[ELOOP]**: 在解析路径名参数过程中遇到符号链接循环。

- **[EMFILE]**: 进程可用的所有文件描述符当前都已打开。
- **[EMFILE]**: {STREAM_MAX} 流在调用进程中当前已打开。
- **[ENAMETOOLONG]**: 路径名的长度超过 {PATH_MAX}，或符号链接的路径名解析产生的中间结果长度超过 {PATH_MAX}。
- **[ENFILE]**: 系统中当前打开的文件数已达到最大允许数量。
- **[ENOENT]**: mode 字符串以 'r' 开头且路径名的某个组件不命名现有文件，或 mode 以 'w' 或 'a' 开头且路径名的路径前缀的某个组件不命名现有文件，或路径名为空字符串。
- **[ENOENT] 或 [ENOTDIR]**: 路径名参数包含至少一个非\ 字符并以一个或多个尾随\ 字符结尾。如果去掉尾随\ 字符的路径名会命名现有文件，则不应发生 [ENOENT] 错误。
- **[ENOSPC]**: 将包含新文件的目录或文件系统无法扩展，文件不存在，且文件将要被创建。
- **[ENOTDIR]**: 路径前缀的某个组件命名既不是目录也不是指向目录的符号链接的现有文件，或路径名参数包含至少一个非\ 字符并以一个或多个尾随\ 字符结尾且最后一个路径名组件命名既不是目录也不是指向目录的符号链接的现有文件。
- **[ENXIO]**: 命名文件是字符特殊文件或块特殊文件，与此特殊文件关联的设备不存在。
- **[EOVERFLOW]**: 命名文件是常规文件且文件大小无法在 off_t 类型的对象中正确表示。
- **[EROFS]**: 命名文件驻留在只读文件系统上且 mode 要求写访问。

`fopen()` 函数可能在以下情况下失败：

- **[EINVAL]**: mode 参数的值无效。
- **[ELOOP]**: 在解析路径名参数过程中遇到超过 {SYMLOOP_MAX} 个符号链接。
- **[EMFILE]**: {FOPEN_MAX} 流在调用进程中当前已打开。
- **[ENAMETOOLONG]**: 路径名的某个组件长度超过 {NAME_MAX}。
- **[ENOMEM]**: 可用存储空间不足。
- **[ETXTBSY]**: 文件是正在执行的纯过程（共享文本）文件且 mode 要求写访问。

示例

打开文件

以下示例尝试打开名为 file 的文件进行读取。 `fopen()` 函数返回一个文件指针，用于后续的 `fgets()` 和 `fclose()` 调用。如果程序无法打开文件，它只会忽略它。

```
#include <stdio.h>
...
FILE *fp;
...
void rgrep(const char *file)
{
...
    if ((fp = fopen(file, "r")) == NULL)
        return;
...
}
```

应用程序用法

如果应用程序需要以文件已存在时失败的方式创建文件，并且要么不需要文件的独占访问，要么不需要独占访问，它应该使用带有 `O_CREAT` 和 `O_EXCL` 标志的 `open()`，而不是使用 mode 中带有 'x' 的 `fopen()`。然后，如果需要，可以通过在 `open()` 返回的文件描述符上调用 `fdopen()` 来创建流。

原理

'e' 模式字符的提供是为了避免多线程应用程序中的数据竞争。没有它，文件描述符会在一个线程创建文件描述符的窗口和另一个线程使用 `fileno()` 和 `fcntl()` 设置 `FD_CLOEXEC` 标志之间泄漏到子进程中。也可以通过使用带有 `O_CLOEXEC` 的 `open()` 然后 `fdopen()` 来避免竞争，但是，`freopen()` 函数没有安全的替代方案，并且一致性要求 'e' 修饰符应该为所有接受模式字符串的函数标准化。

ISO C 标准只识别模式字符串中特定位置的 '+'、'b' 和 'x' 字符，将其他安排保留为未指定，并且只允许 'x' 在以 'w' 开头的模式字符串中。本标准特别要求支持模式字符串中除第一个字符外的所有字符以任何顺序识别。因此，"wxe" 和 "wex" 行为相同，虽然 "wx+" 在 ISO C 标准中未指定，但本标准要求它与

"w+x" 具有相同的行为。本标准还要求 'x' 适用于以 'a' 开头的模式字符串，以及对于以 'r' 开头的模式字符串具有实现定义的行为。因此，虽然 `open()` 在指定 `O_EXCL` 而不指定 `O_CREAT` 时具有未定义行为，但 `fopen()` 并非如此。

当 'x' 在 mode 中时，ISO C 标准要求文件在底层系统支持独占访问的范围内以独占访问创建。虽然 POSIX.1 没有指定启用独占访问的任何方法，但它允许存在实现特定的标志或标志来启用它。请注意，如果正在创建文件，它们应该是文件创建标志，而不是文件访问模式标志（即包含在 `O_ACCMODE` 中的标志）或文件状态标志，这样它们就不会影响 `fcntl()` 使用 `F_GETFL` 返回的值。在具有此类标志的实现上，如果对它们的支持依赖于文件系统，并且在 `fopen()` 用于在不支持它的文件系统上创建文件时请求独占访问，则如果这些标志会导致 `fopen()` 失败，则不得使用它们。

一些实现支持强制性文件锁定作为启用文件独占访问的手段。锁以正常方式设置，但它们不仅防止其他人设置冲突锁，还防止其他人以与锁冲突的方式访问文件的锁定部分的内容。但是，除非实现在文件创建时有一种设置整个文件写锁的方法，否则这不能满足 ISO C 标准中文件"在底层系统支持独占访问的范围内以独占访问创建"的要求。（让 `fopen()` 创建文件并在文件上设置锁作为两个独立的操作是不同的，并且它会引入竞争条件，即另一个进程可以在两个操作之间打开文件并写入它（或设置锁）。）然而，在所有支持强制性文件锁定的实现上，不鼓励使用它；因此，建议支持强制性文件锁定的实现不要添加创建具有整个文件独占锁的文件的方法，这样 `fopen()` 就不需要启用强制性文件锁定以符合 ISO C 标准。具有创建具有整个文件独占锁的文件方法的实现需要提供一种根据调用进程是在 POSIX.1 符合环境还是 ISO C 符合环境中执行来改变 `fopen()` 行为的方法。

模式 "rx" 的典型实现定义行为是忽略 'x'，但标准开发者不希望强制要求这种行为。例如，实现可以允许共享访问进行读取；即，不允许以这种方式打开的文件也被打开进行写入。

鼓励实现在 mode 以 'w' 或 'a' 开头、文件先前不存在且路径名的最后一个组件包含任何具有 \ 字符编码值的字节时，让 `fopen()` 和 `freopen()` 报告 [EILSEQ] 错误。

未来方向

无。

另请参阅

- 2.5 标准 I/O 流

- `creat()`
- `fclose()`
- `fdopen()`
- `fmemopen()`
- `freopen()`
- `open_memstream()`
-

更改历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

添加了大文件峰会扩展。

Issue 6

超出 ISO C 标准的扩展被标记。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 DESCRIPTION 中，添加了文本以指示在打开文件描述中设置偏移量最大值。此更改是为了支持大文件。
- 在 ERRORS 部分，添加了 [EOVERFLOW] 条件。此更改是为了支持大文件。
- 添加了 [ELOOP] 强制错误条件。
- 添加了 [EINVAL]、[EMFILE]、[ENAMETOOLONG]、[ENOMEM] 和 [ETXTBSY] 可选错误条件。

规范性文本已更新，以避免对应用程序要求使用"必须"一词。

为与 ISO/IEC 9899:1999 标准对齐进行以下更改：

- `fopen()` 的原型已更新。
- DESCRIPTION 已更新，以注意如果参数 mode 指向的字符串不是列出的那些，则行为未定义。

强制性 [ELOOP] 错误条件的措辞已更新，并添加了第二个可选 [ELOOP] 错误条件。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #025，澄清文件创建模式。

应用 Austin Group Interpretation 1003.1-2001 #143。

应用 Austin Group Interpretation 1003.1-2001 #159，澄清在打开文件描述上设置的标志要求。

应用 SD5-XBD-ERN-4，更改 [EMFILE] 错误的定义。

应用 SD5-XSH-ERN-149，将 {STREAM_MAX} [EMFILE] 错误条件从"可能失败"更改为"应失败"。

进行了与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0156 [291,433]、XSH/TC1-2008/0157 [146,433]、XSH/TC1-2008/0158 [324] 和 XSH/TC1-2008/0159 [14]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0122 [822]。

Issue 8

应用 Austin Group Defect 251，鼓励实现禁止创建包含任何具有 \ 字符编码值的字节的文件名。

应用 Austin Group Defect 293，添加 [EILSEQ] 错误。

应用 Austin Group Defects 411 和 1524，添加 'e' 和 'x' 模式字符串字符。

应用 Austin Group Defect 1200，更正 [ELOOP] 错误中的参数名称。

应用 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。

1.46. fprintf

概要

```
#include <stdio.h>

/* [CX] */ int asprintf(char **restrict ptr, const char *restrict
/* [CX] */ int dprintf(int fildes, const char *restrict format,
int fprintf(FILE *restrict stream, const char *restrict format,
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
    const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...)
```

描述

/ [CX] / 除了 asprintf()、dprintf() 函数以及当传递空宽字符时 %lc 转换的行为外，本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何其他冲突都是非预期的。本卷 POSIX.1-2024 对于所有 fprintf()、printf()、snprintf() 和 sprintf() 功能都遵从 ISO C 标准，除了当传递空宽字符时的 %lc 转换。

fprintf() 函数应将输出放置到指定的输出流上。printf() 函数应将输出放置到标准输出流 stdout 上。sprintf() 函数应将输出后随空字节 '\0' 放置到从 *s 开始的连续字节中；用户有责任确保有足够的空间可用。

/ [CX] / asprintf() 函数应等同于 sprintf()，不同之处在于输出字符串应写入动态分配的内存中，该内存通过相当于调用 malloc() 的方式分配，长度足以容纳结果字符串，包括终止空字节。如果 asprintf() 调用成功，此动态分配字符串的地址应存储在 ptr 引用的位置。

/ [CX] / dprintf() 函数应等同于 fprintf() 函数，不同之处在于 dprintf() 应将输出写入与 fildes 参数指定的文件描述符关联的文件，而不是将输出放置到流上。

snprintf() 函数应等同于 sprintf()，增加了 n 参数，该参数限制写入到 s 引用的缓冲区的字节数。如果 n 为零，则不应写入任何内容，s 可以为空指针。否则，超出第 n-1 个字节的输出字节将被丢弃而不是写入数组，并且在实际写入数组的字节末尾写入一个空字节。

如果由于调用 sprintf() 或 snprintf() 而在重叠的对象之间进行复制，则结果是未定义的。

这些函数中的每一个都在格式控制下转换、格式化并打印其参数。应用程序应确保格式是一个字符串，如果有移位状态，则以其初始移位状态开始和结束。格式由零个或多个指令组成：普通字符（被简单地复制到输出流）和转换规范（每个都将导致获取零个或多个参数）。如果格式没有足够的参数，则结果是未定义的。如果格式用尽但参数仍有剩余，则多余的参数应被求值，但在其他情况下被忽略。

/ [CX] / 转换可以应用于参数列表中格式之后的第 n 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 '%'（见下文）被序列 "%n\$" 替换，其中 n 是范围 [1,{NL_ARGMAX}] 内的十进制整数，给出参数在参数列表中的位置。此功能提供了定义格式字符串的能力，这些字符串可以选择适合特定语言的参数顺序（见示例部分）。

格式可以包含编号参数转换规范（即由 "%n\$" 引入的规范，可选包含 "*m\$" 形式的字段宽度和精度）或未编号参数转换规范（即由 % 字符引入的规范，可选包含 * 形式的字段宽度和精度），但不能同时包含两者。唯一的例外是 %% 可以与 "%n\$" 形式混合使用。在格式字符串中混合编号和未编号参数规范的结果是未定义的。使用编号参数规范时，指定第 N 个参数要求所有前导参数（从第一个到第 N-1 个）都在格式字符串中指定。

/ [CX] / 在包含 "%n\$" 形式转换规范的格式字符串中，参数列表中的编号参数可以从格式字符串中根据需要引用多次。

在包含 % 形式转换规范的格式字符串中，每个转换规范使用参数列表中第一个未使用的参数。

/ [CX] / 所有形式的 `fprintf()` 函数都允许在输出字符串中插入依赖于语言的基数字符。基数字符在当前语言环境（类别 LC_NUMERIC）中定义。在 POSIX 语言环境中，或在未定义基数字符的语言环境中，基数字符应默认为 ('.')。

每个转换规范由 '%' 字符 */ [CX]* / 或字符序列 "%n\$" 引入，之后按顺序出现以下内容：

- 零个或多个标志（按任意顺序），它们修改转换规范的含义。
- 可选的最小字段宽度。如果转换后的值具有比字段宽度更少的字节，则默认情况下应在左侧用字符填充；如果字段宽度被赋予左调整标志（'-'）（如下所述），则应在右侧填充。字段宽度采用('') 的形式，*/ [CX]* / 或在由 "%n\$" 引入的转换规范中采用下面描述的 "m\$" 字符串，或十进制整数的形式。
- 可选的精度，它给出要出现的最小数字数，用于 d、i、o、u、x 和 X 转换说明符；要出现在基数字符之后的数字数，用于 a、A、e、E、f 和 F 转换说明符；最大有效数字数，用于 g 和 G 转换说明符；或要从字符串中打印的最大字节数，用于 s */ [XSI]* / 和 S 转换说明符。精度采用('') 后随('') 的形式，*/ [CX]* / 或在由 "%n\$" 引入的转换规范中采用下面描述的 "m\$" 字符

串, 或可选的十进制数字字符串的形式, 其中空数字字符串被视为零。如果精度与任何其他转换说明符一起出现, 则行为是未定义的。

- 可选的长度修饰符, 它指定参数的大小。
- 转换说明符字符, 指示要应用的转换类型。

字段宽度或精度或两者都可以由(')指示。在这种情况下, *int*类型的参数提供字段宽度或精度。应用程序应确保指定字段宽度或精度或两者的参数按该顺序出现在要转换的参数(如果有)之前。负字段宽度被视为'-'标志后跟正字段宽度。负精度被视为精度被省略。/[CX]/在包含由"%n\$"引入的转换规范的格式字符串中, 除了由十进制数字字符串指示外, 字段宽度可以由序列"m\$"指示, 精度可以由序列".*m\$"指示, 其中m是范围[1,{NL_ARGMAX}]内的十进制整数, 给出包含字段宽度或精度的整数参数在参数列表中的位置(在格式参数之后), 例如:

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

标志字符

标志字符及其含义是:

'(撇号)

/ [CX]/十进制转换(%i、%d、%u、%f、%F、%g或%G)的结果的整数部分应使用千位分组字符进行格式化。对于其他转换, 行为是未定义的。使用非货币分组字符。

-

转换结果应在字段内左对齐。如果未指定此标志, 则转换是右对齐的。

+

有符号转换的结果应始终以符号('+'或'-)开头。如果未指定此标志, 则仅当转换负值时转换才以符号开头。

如果有符号转换的第一个字符不是符号, 或者如果有符号转换没有产生字符, 则应在结果前添加。这意味着如果和'+'标志都出现, 则标志应被忽略。

#

指定值将被转换为替代形式。对于o转换, 它应增加精度, 当且仅当必要时, 强制结果的第一个数字为零(如果值和精度都为0, 则打印单个0)。对于x或X转换说明符, 非零结果应前缀0x(或0X)。对于a、A、e、E、f、F、g和G转换说明符, 结果应始终包含基数字符, 即使基数字符后没有数字。没有此标志, 基数字符仅在其后有数字时才出现在这些转换的结果中。对于g和G转换

说明符，尾随零不应像通常那样从结果中移除。对于其他转换说明符，行为是未定义的。

0

对于 d、i、o、u、x、X、a、A、e、E、f、F、g 和 G 转换说明符，使用前导零（在任何符号或基数指示之后）来填充到字段宽度，而不是执行空格填充，除非转换无穷大或 NaN。如果 '0' 和 '-' 标志都出现，则忽略 '0' 标志。对于 d、i、o、u、x 和 X 转换说明符，如果指定了精度，则忽略 '0' 标志。/ [CX] / 如果 '0' 和 标志都出现，则分组字符在零填充之前插入。对于其他转换，行为是未定义的。

长度修饰符

长度修饰符及其含义是：

hh

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 signed char 或 unsigned char 参数（参数将根据整数提升进行提升，但在打印前其值应转换为 signed char 或 unsigned char）；或指定后续的 n 转换说明符应用于指向 signed char 参数的指针。

h

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 short 或 unsigned short 参数（参数将根据整数提升进行提升，但在打印前其值应转换为 short 或 unsigned short）；或指定后续的 n 转换说明符应用于指向 short 参数的指针。

l (ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 long 或 unsigned long 参数；指定后续的 n 转换说明符应用于指向 long 参数的指针；指定后续的 c 转换说明符应用于 wint_t 参数；指定后续的 s 转换说明符应用于指向 wchar_t 参数的指针；或对后续的 a、A、e、E、f、F、g 或 G 转换说明符没有影响。

ll (ell-ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 long long 或 unsigned long long 参数；或指定后续的 n 转换说明符应用于指向 long long 参数的指针。

j

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 intmax_t 或 uintmax_t 参数；或指定后续的 n 转换说明符应用于指向 intmax_t 参数的指针。

z

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 size_t 或相应的有符号整数类型参数；或指定后续的 n 转换说明符应用于指向对应 size_t 参数的有符号整数类型的指针。

t

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 ptrdiff_t 或相应的无符号类型参数；或指定后续的 n 转换说明符应用于指向 ptrdiff_t 参数的指针。

L

指定后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于 long double 参数。

如果长度修饰符与除上述指定之外的任何转换说明符一起出现，则行为是未定义的。

转换说明符

转换说明符及其含义是：

d, i

int 参数应转换为有符号十进制格式 "[-]dddd"。精度指定要出现的最小数字数；如果要转换的值可以用更少的数字表示，则应使用前导零进行扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

o

unsigned 参数应转换为无符号八进制格式 "dddd"。精度指定要出现的最小数字数；如果要转换的值可以用更少的数字表示，则应使用前导零进行扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

u

unsigned 参数应转换为无符号十进制格式 "dddd"。精度指定要出现的最小数字数；如果要转换的值可以用更少的数字表示，则应使用前导零进行扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

x

unsigned 参数应转换为无符号十六进制格式 "dddd"；使用字母 "abcdef"。精度指定要出现的最小数字数；如果要转换的值可以用更少的数字表示，则应使用前导零进行扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

X

等同于 x 转换说明符，不同之处在于使用字母 "ABCDEF" 而不是 "abcdef"。

f, F

double 参数应转换为十进制表示法 "[-]ddd.ddd"，其中基数字符后的数字数等于精度规范。如果缺少精度，则应取为 6；如果精度显式为零且没有 '#' 标志，则不应出现基数字符。如果出现基数字符，则其前面至少有一个数字。低位数字应以实现定义的方式进行四舍五入。

表示无穷大的 double 参数应转换为 "[-]inf" 或 "[-]infinity" 样式中的一种；使用哪种样式是实现定义的。表示 NaN 的 double 参数应转换为 "[-]nan(n-character-sequence)" 或 "[-]nan" 样式中的一种；使用哪种样式以及任何 n-character-

sequence 的含义是实现定义的。F 转换说明符产生 "INF"、"INFINITY" 或 "NAN" 而不是 "inf"、"infinity" 或 "nan"。

e, E

double 参数应转换为 "[-]d.ddd e \pm dd" 样式，其中基数字符前有一个数字（如果参数非零则为非零），其后的数字数等于精度；如果缺少精度，则应取为 6；如果精度为零且没有 '#' 标志，则不应出现基数字符。低位数字应以实现定义的方式进行四舍五入。E 转换说明符应产生用 'E' 而不是 'e' 引入指数的数字。指数应始终包含至少两个数字。如果值为零，则指数应为零。

表示无穷大或 NaN 的 double 参数应转换为 f 或 F 转换说明符的样式。

g, G

表示浮点数的 double 参数应根据转换的值和精度转换为 f 或 e 样式（或在 G 转换说明符的情况下转换为 F 或 E 样式）。令 P 等于精度（如果非零）、如果精度省略则为 6、如果精度为零则为 1。然后，如果使用 E 样式的转换具有指数 X：

- 如果 $P > X \geq -4$ ，则转换应使用 f（或 F）样式和精度 $P-(X+1)$ 。
- 否则，转换应使用 e（或 E）样式和精度 P-1。

最后，除非使用 '#' 标志，否则任何尾随零都应从结果的小数部分中移除，如果没有剩余的小数部分，则应移除小数点字符。

表示无穷大或 NaN 的 double 参数应转换为 f 或 F 转换说明符的样式。

a, A

表示浮点数的 double 参数应转换为 "[-]0x h.hhhh p \pm d" 样式，其中基数字符前有一个十六进制数字（如果参数是标准化的浮点数则应为非零，否则未指定），其后的十六进制数字数等于精度；如果缺少精度且 FLT_RADIX 是 2 的幂，则精度应足以精确表示该值；如果缺少精度且 FLT_RADIX 不是 2 的幂，则精度应足以区分 double 类型的值，但可以省略尾随零；如果精度为零且未指定 '#' 标志，则不应出现小数点字符。对于 a 转换使用字母 "abcdef"，对于 A 转换使用字母 "ABCDEF"。A 转换说明符产生用 'X' 和 'P' 而不是 'x' 和 'p' 的数字。指数应始终包含至少一个数字，并且只包含表示 2 的十进制指数所需的额外数字。如果值为零，则指数应为零。

表示无穷大或 NaN 的 double 参数应转换为 f 或 F 转换说明符的样式。

c

int 参数应转换为 unsigned char，并将结果字节写入。

如果存在 l (ell) 限定符，/ [CX] / wint_t 参数应转换为多字节序列，如同通过调用 wcrtomb() 并使用指向至少 MB_CUR_MAX 字节存储的指针、转换为 wchar_t 的 wint_t 参数和初始移位状态，并将结果字节写入。

s

参数应是指向 char 数组的指针。应从数组中写入字节直到（但不包括）任何终

止空字节。如果指定了精度，则写入的字节数不应超过该数量。如果未指定精度或精度大于数组的大小，应用程序应确保数组包含空字节。

如果存在 l (ell) 限定符，参数应是指向 wchar_t 类型数组的指针。数组中的宽字符应转换为字符（每个都如同通过调用 wcrtomb() 函数，转换状态由在第一个宽字符转换前初始化为零的 mbstate_t 对象描述），直到并包括终止空宽字符。应写入结果字符直到（但不包括）终止空字符（字节）。如果未指定精度，应用程序应确保数组包含空宽字符。如果指定了精度，则写入的字符（字节）数不应超过该数量（包括任何移位序列），并且如果函数需要访问数组末尾之后的宽字符以等于精度给定的字符序列长度，则数组应包含空宽字符。在任何情况下都不应写入部分字符。

p

参数应是指向 void 的指针。指针的值以实现定义的方式转换为可打印字符序列。

n

参数应是指向整数的指针，其中写入到目前为止通过调用 fprintf() 函数之一写入输出的字节数。不转换任何参数。

c

/ [XSI] / 等同于 lc。

s

/ [XSI] / 等同于 ls。

%

写入 '%' 字符；不应转换任何参数。应用程序应确保完整的转换规范是 %%。

如果转换规范与上述形式之一不匹配，则行为是未定义的。如果任何参数不是相应转换规范的正确类型，则行为是未定义的。

在任何情况下，不存在或过小的字段宽度都不会导致字段截断；如果转换的结果比字段宽度更宽，则应扩展字段以包含转换结果。由 fprintf() 和 printf() 生成的字符被打印，如同调用了 fputc()。

对于 a 和 A 转换说明符，如果 FLT_RADIX 是 2 的幂，则值应正确四舍五入为具有给定精度的十六进制浮点数。

对于 a 和 A 转换，如果 FLT_RADIX 不是 2 的幂且结果不能用给定精度精确表示，则结果应为具有给定精度的十六进制浮点样式的两个相邻数字之一，额外要求误差对于当前舍入方向应有正确的符号。

对于 e、E、f、F、g 和 G 转换说明符，如果有效十进制数字的数量最多为 DECIMAL_DIG，则结果应正确四舍五入。如果有效十进制数字的数量超过 DECIMAL_DIG 但源值可以用 DECIMAL_DIG 位数字精确表示，则结果应为带尾随零的精确表示。否则，源值由两个相邻的十进制字符串 L < U 界定，两者都具

有 DECIMAL_DIG 位有效数字；结果十进制字符串 D 的值应满足 $L \leq D \leq U$ ，
额外要求误差对于当前舍入方向应有正确的符号。

/ [CX] / 文件的最后数据修改和最后文件状态更改时间戳应标记为更新：

1. 在调用成功执行 `fprintf()` 或 `printf()` 与在同一流上成功完成下一次调用 `fflush()` 或 `fclose()` 调用或调用 `exit()` 或 `abort()` 之间
2. 成功完成调用 `dprintf()` 时

返回值

成功完成时，/ [CX] / `dprintf()`、`fprintf()` 和 `printf()` 函数应返回传输的字节数。

/ [CX] / 成功完成时，`asprintf()` 函数应返回写入存储在 `ptr` 引用位置的分配字符串的字节数，不包括终止空字节。

成功完成时，`sprintf()` 函数应返回写入 `s` 的字节数，不包括终止空字节。

成功完成时，`snprintf()` 函数应返回如果 `n` 足够大则应写入 `s` 的字节数，不包括终止空字节。

如果遇到错误，这些函数应返回负值 / [CX] / 并设置 `errno` 以指示错误。对于 `asprintf()`，如果内存分配不可能，或者发生其他错误，函数应返回负值，并且 `ptr` 引用的位置的内容是未定义的，但不应引用已分配的内存。

如果在调用 `snprintf()` 时 `n` 的值为零，则不应写入任何内容，应返回如果 `n` 足够大则应写入的字节数（不包括终止空字节），并且 `s` 可以为空指针。

错误

关于 / [CX] / `dprintf()`、`fprintf()` 和 `printf()` 失败和可能失败的条件，请参考 `fputc()` 或 `fputwc()`。

此外，所有形式的 `fprintf()` 在以下情况下应失败：

[EILSEQ]

/ [CX] / 检测到不对应于有效字符的宽字符代码。

[EOVERFLOW]

/ [CX] / 要返回的值大于 {INT_MAX}。

/ [CX] / `asprintf()` 函数在以下情况下应失败：

[ENOMEM]

存储空间不足。

/ [CX] / `dprintf()` 函数在以下情况下可能失败：

[EBADF]

fildes 参数不是有效的文件描述符。

/ [CX] / dprintf()、fprintf() 和 printf() 函数在以下情况下可能失败：

[ENOMEM]

/ [CX] / 存储空间不足。

以下部分是提供信息的。

示例

打印与语言无关的日期和时间

以下语句可用于使用与语言无关的格式打印日期和时间：

```
printf(format, weekday, month, day, hour, min);
```

对于美国用法，format 可以是指向以下字符串的指针：

```
"%s, %s %d, %d:%.2d\n"
```

此示例将产生以下消息：

```
Sunday, July 3, 10:02
```

对于德国用法，format 可以是指向以下字符串的指针：

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

此 format 定义将产生以下消息：

```
Sonntag, 3. Juli, 10:02
```

打印文件信息

以下示例打印目录中特定文件的类型、权限和链接数信息。

前两次对 printf() 的调用使用从先前 stat() 调用解码的数据。用户定义的 strperm() 函数应返回类似于以下命令输出开头的字符串：

```
ls -l
```

对 `printf()` 的下一次调用输出所有者名称（如果使用 `getpwuid()` 找到）；`getpwuid()` 函数应返回 `passwd` 结构，从中提取用户名。如果未找到用户名，程序则打印用户 ID 的数值。

下一次调用打印组名（如果使用 `getgrgid()` 找到）；`getgrgid()` 与 `getpwuid()` 非常相似，不同之处在于它应基于组号返回组信息。同样，如果未找到组，程序打印该条目的组数值。

对 `printf()` 的最后一次调用打印文件的大小。

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);

...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...

printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
...
```

打印本地化日期字符串

以下示例获取本地化日期字符串。`nl_langinfo()` 函数应返回本地化日期字符串，该字符串指定日期的顺序和布局。`strftime()` 函数接受此信息，并使用 `tm` 结构中的值，将日期和时间信息放入 `datestring` 中。然后 `printf()` 函数输出 `datestring` 和条目的名称。

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT),
printf(" %s %s\n", datestring, dp->d_name);
...
```

打印错误信息

以下示例使用 `fprintf()` 将错误信息写入标准错误。

在第一组调用中，程序尝试打开名为 `LOCKFILE` 的密码锁定文件。如果文件已存在，则这是一个错误，如 `open()` 函数上的 `O_EXCL` 标志所示。如果调用失败，程序假设其他人正在更新密码文件，程序退出。

下一组调用通过在 `LOCKFILE` 和新密码文件 `PASSWDFILE` 之间创建链接，将新密码文件保存为当前密码文件。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n"
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
```

```
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
...
```

打印使用信息

以下示例检查程序是否有必要的参数，如果预期的参数数量不存在，则使用 `fprintf()` 打印使用信息。

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file>\n", argv[0], Options)
}
...

```

格式化十进制字符串

以下示例在 `stdout` 上打印键和数据对。注意格式字符串中 ('*') 的使用；这确保基于请求的元素数量为元素提供正确的小数位数。

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

创建路径名

以下示例使用先前返回用户密码数据库条目的 `getpwnam()` 函数的信息创建路径名。

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
...
char *pathname;
struct passwd *pw;
size_t len;
...
// pid_t 所需的数字是位数乘以
// log2(10) = 约等于 10/33
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +
      sizeof ".out";
pathname = malloc(len);
if (pathname != NULL)
{
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,
             (intmax_t)getpid());
    ...
}
```

报告事件

以下示例循环直到事件超时。`pause()` 函数永远等待，除非收到信号。由于 `pause()` 的可能返回值，`fprintf()` 语句不应该发生。

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
    ...
    if (pause() != -1 || errno != EINTR)
        fprintf(stderr, "pause: unknown error: %s\n", strerror(
    }
    ...

```

打印货币信息

以下示例使用 `strfmon()` 转换数字并将其存储为名为 `convbuf` 的格式化货币字符串。如果打印第一个数字，程序打印格式和描述；否则，它只打印数字。

```

#include <monetary.h>
#include <stdio.h>
...
struct tblfmt {
    char *format;
    char *description;
};

struct tblfmt table[] = {
    { "%n", "默认格式化" },
    { "%11n", "在 11 个字符字段内右对齐" },
    { "%#5n", "对齐最大 99999 值的列" },
    { "%=*#5n", "指定填充字符" },
    { "%=0#5n", "填充字符不使用分组" },
    { "%^#5n", "禁用分组分隔符" },
    { "%^#5.0n", "四舍五入到整数单位" },
    { "%^#5.4n", "增加精度" },
    { "%(#5n", "使用替代正/负样式" },
    { "%!(#5n", "禁用货币符号" },
};

float input[3];
int i, j;
char convbuf[100];
...
strftime(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s %s %s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf(" %s\n", convbuf);
}
...

```

打印宽字符

以下示例打印一系列宽字符。假设 "L @" 扩展为三个字节：

```

wchar_t wz [3] = L"@@";           // 零终止
wchar_t wn [3] = L"@@";           // 非终止

fprintf (stdout,"%ls", wz);      // 输出 6 字节
fprintf (stdout,"%ls", wn);      // 未定义, 因为 wn 没有终止符
fprintf (stdout,"%4ls", wz);     // 输出 3 字节
fprintf (stdout,"%4ls", wn);     // 输出 3 字节; 不需要终止符

```

```
fprintf (stdout, "%9ls", wz); // 输出 6 字节
fprintf (stdout, "%9ls", wn); // 输出 9 字节; 不需要终止符
fprintf (stdout, "%10ls", wz); // 输出 6 字节
fprintf (stdout, "%10ls", wn); // 未定义, 因为 wn 没有终止符
```

在示例的最后一行, 处理三个字符后, 已输出九个字节。然后必须检查第四个字符以确定它是否转换为一个或更多字节。如果它转换为多于一个字节, 则输出只有九个字节。由于数组中没有第四个字符, 行为是未定义的。

应用程序用法

如果调用 `fprintf()` 的应用程序具有任何 `wint_t` 或 `wchar_t` 类型的对象, 它还必须包含头文件以定义这些对象。

成功调用 `asprintf()` 分配的空间应随后通过调用 `free()` 释放。

基本原理

如果实现检测到格式没有足够的参数, 建议函数应失败并报告 `[EINVAL]` 错误。

为 `%lc` 转换指定的行为与 ISO C 标准中的规范略有不同, 因为打印空宽字符会产生空字节而不是产生 0 字节输出 (正如严格阅读 ISO C 标准关于表现得像对第一个元素是空宽字符的 `wchar_t` 数组应用 `%ls` 说明符的指示所要求的那样)。要求为每个可能的宽字符 (包括空字符) 提供多字节输出, 这符合历史实践, 并提供了与 `fprintf()` 中的 `%c` 以及 `fwprintf()` 中的 `%c` 和 `%lc` 的一致性。预计 ISO C 标准的未来版本将更改以匹配此处指定的行为。

未来方向

无。

参见

`2.5 标准 I/O 流`, `fputc()`, `fscanf()`, `setlocale()`, `strfmon()`, `strlcat()`, `wcrtomb()`, `wcslcat()`

`XBD 7. 语言环境`, , ,

更改历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

与 ISO/IEC 9899:1990/Amendment 1:1995 (E) 对齐。具体而言，l (ell) 限定符现在可以与 c 和 s 转换说明符一起使用。

snprintf() 函数在 Issue 5 中新增。

Issue 6

超出 ISO C 标准的扩展被标记。

规范文本已更新，以避免对应用程序要求使用术语"必须"。

为与 ISO/IEC 9899:1999 标准对齐进行以下更改：

- fprintf()、printf()、snprintf() 和 sprintf() 的原型已更新，snprintf() 的 XBI 阴影被移除。
- snprintf() 的描述与 ISO C 标准对齐。请注意，这取代了 The Open Group Base Resolution bwg98-006 中的 snprintf() 描述，该描述改变了 Issue 5 的行为。
- DESCRIPTION 已更新。

DESCRIPTION 已更新，以一致使用术语"转换说明符"和"转换规范"。

纳入了 ISO/IEC 9899:1999 标准，技术勘误 1。

添加了打印宽字符的示例。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #161，更新 0 标志的 DESCRIPTION。

应用 Austin Group Interpretation 1003.1-2001 #170。

应用 ISO/IEC 9899:1999 标准，技术勘误 2 #68 (SD5-XSH-ERN-70)，修订 g 和 G 的描述。

应用 SD5-XSH-ERN-174。

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 添加 dprintf() 函数。

与 %n\$ 形式转换规范和 标志相关的功能从 XBI 选项移到 Base。

进行与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0163 [302], XSH/TC1-2008/0164 [316], XSH/TC1-2008/0165 [316], XSH/TC1-2008/0166 [451,291], 和 XSH/TC1-2008/0167 [14]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0126 [894], XSH/TC2-2008/0127 [557], 和 XSH/TC2-2008/0128 [936]。

Issue 8

应用 Austin Group Defect 986, 将 strlcat() 和 wcslcat() 添加到 SEE ALSO 部分。

应用 Austin Group Defect 1020, 阐明 snprintf() 参数 n 限制写入 s 的字节数; 它不一定与 s 的大小相同。

应用 Austin Group Defect 1021, 在 RETURN VALUE 部分将"输出错误"更改为"错误"。

应用 Austin Group Defect 1137, 阐明在转换规范中使用 "%n\$" 和 "*m\$"。

应用 Austin Group Defect 1205, 更改 % 转换说明符的描述。

应用 Austin Group Defect 1219, 移除 snprintf() 特定的 [EOVERFLOW] 错误。

应用 Austin Group Defect 1496, 添加 asprintf() 函数。

应用 Austin Group Defect 1562, 阐明应用程序有责任确保格式是一个字符串, 如果有移位状态, 则以其初始移位状态开始和结束。

应用 Austin Group Defect 1647, 更改 c 转换说明符的描述, 并更新本卷 POSIX.1-2024 遵从 ISO C 标准的陈述, 以便其排除传递空宽字符时的 %lc 转换。

1.47. fputc — 向流写入一个字节

概要

```
#include <stdio.h>

int fputc(int c, FILE *stream);
```

描述

`fputc()` 函数应将由 `c` 指定的字节（转换为 `unsigned char` 类型）写入到 `stream` 指向的输出流中，写入位置为该流相关文件位置指示器所指示的位置（如果已定义），并应适当地推进指示器。如果文件不支持定位请求，或者流是以追加模式打开的，则字节应被追加到输出流中。

文件的数据最后修改时间戳和文件状态最后更改时间戳应在 `fputc()` 成功执行与下一次对同一流成功完成 `fflush()` 或 `fclose()` 调用，或调用 `exit()` 或 `abort()` 之间被标记为需要更新。

返回值

成功完成后，`fputc()` 应返回其写入的值。否则，应返回 EOF，流的错误指示器应被设置，且 `errno` 应被设置为指示错误。

错误

如果流是无缓冲的，或者流的缓冲区需要刷新，则 `fputc()` 函数可能失败：

- **[EAGAIN]**

为 `stream` 底层的文件描述符设置了 `O_NONBLOCK` 标志，且线程在写操作中会被延迟。

- **[EBADF]**

`stream` 底层的文件描述符不是有效的、以写入方式打开的文件描述符。

- **[EFBIG]**

尝试写入的文件超过了最大文件大小。

- **[EFBIG]**

尝试写入的文件超过了进程的文件大小限制。同时还应为线程生成 SIGXFSZ 信号。

- **[EFBIG]**

文件是常规文件，且尝试写入的位置达到或超过了偏移量最大值。

- **[EINTR]**

写操作因接收到信号而终止，且没有数据被传输。

- **[EIO]**

发生物理 I/O 错误，或者进程是后台进程组的成员，尝试写入其控制终端，TOSTOP 被设置，调用线程未阻塞 SIGTTOU，进程未忽略 SIGTTOU，且进程的进程组是孤立的。在实现定义的条件下也可能返回此错误。

- **[ENOSPC]**

包含文件的设备上没有剩余的可用空间。

- **[EPIPE]**

尝试写入管道或 FIFO，但没有进程为读取而打开。同时还应向线程发送 SIGPIPE 信号。

`fputc()` 函数在以下情况下可能失败：

- **[ENOMEM]**

可用存储空间不足。

- **[ENXIO]**

请求了不存在的设备，或者请求超出了设备的能力。

示例

无。

应用用法

无。

基本原理

无。

未来方向

无。

参见

2.5 标准 I/O 流, `ferror()`, `fopen()`, `getrlimit()`, `putc()`,
`puts()`, `setbuf()`

XBD `<stdio.h>`

变更历史

首次发布于 Issue 1。派生自 SVID 的 Issue 1。

Issue 5

添加了大文件峰会扩展。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 添加了 [EIO] 和 [EFBIG] 强制错误条件。
- 添加了 [ENOMEM] 和 [ENXIO] 可选错误条件。

应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/37, 将 ERRORS 部分中的 [EAGAIN] 错误从"进程会被延迟"更新为"线程会被延迟"。

Issue 7

进行了与细粒度时间戳支持相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0168 [79] 和 XSH/TC1-2008/0169 [14]。

Issue 8

应用了 Austin Group Defect 308, 阐明了 [EFBIG] 错误的处理。

应用了 Austin Group Defect 1669，移除了与进程文件大小限制相关的 [EFBIG]
错误部分的 XSI 着色。

1.48. fputs

概要

```
#include <stdio.h>

int fputs(const char *restrict s, FILE *restrict stream);
```

描述

`fputs()` 函数应将由 `s` 指向的以 null 结尾的字符串写入由 `stream` 指向的流中。终止的 null 字节不应被写入。

文件的数据最后修改时间戳和文件状态最后更改时间戳应在 `fputs()` 成功执行与下一次在同一流上成功完成 `fflush()` 或 `fclose()` 调用，或调用 `exit()` 或 `abort()` 之间被标记为更新。

返回值

成功完成时，`fputs()` 应返回一个非负数。否则，应返回 EOF，为流设置错误指示符，并设置 `errno` 以指示错误。

错误

参考 `fputc()`。

示例

打印到标准输出

以下示例获取当前时间，使用 `localtime()` 和 `asctime()` 将其转换为字符串，并使用 `fputs()` 将其打印到标准输出。然后打印等待事件的分钟数。

```
#include <time.h>
#include <stdio.h>
...
```

```
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
       minutes_to_event);
...
```

应用用法

`puts()` 函数会附加一个 `<newline>` (换行符)，而 `fputs()` 不会。

POSIX.1-2024 这一卷要求成功完成时仅返回一个非负整数。对于此要求，至少有三种已知的不同的实现约定：

- 返回一个常量值。
- 返回最后写入的字符。
- 返回写入的字节数。请注意，对于长度超过 `{INT_MAX}` 字节的字符串，此实现约定无法遵循，因为该值无法在函数的返回类型中表示。为了向后兼容，对于长度不超过 `{INT_MAX}` 字节的字符串，实现可以返回字节数，对于所有更长的字符串，返回 `{INT_MAX}`。

基本原理

`fputs()` 函数是在引用的 *The C Programming Language* 中指定了源代码的函数之一。在最初版本中，该函数没有定义返回值，但许多实际的实现会作为副作用返回最后写入字符的值，因为这是累加器中剩余的用作返回值的值。在该书的第二版中，将根据 `ferror()` 的返回值返回固定值 0 或 EOF；然而，为了与现有实现兼容，多个实现在成功时会返回一个表示最后写入字节的正值。

未来方向

无。

另请参见

- 2.5 标准 I/O 流

- `fopen()`
- `putc()`
- `puts()`
- XBD `<stdio.h>`

更改历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

更新了 `fputs()` 原型以与 ISO/IEC 9899:1999 标准对齐。

Issue 7

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0170 [174,412], XSH/TC1-2008/0171 [412], 和 XSH/TC1-2008/0172 [14]。

1.49. fread

SYNOPSIS

```
#include <stdio.h>

size_t fread(void *restrict ptr, size_t size, size_t nitems,
             FILE *restrict stream);
```

DESCRIPTION

`fread()` 函数应从 `stream` 所指向的流中读取最多 `nitems` 个元素到 `ptr` 所指向的数组，每个元素的大小由 `size` 字节指定。对于每个对象，应调用 `size` 次 `fgetc()` 函数，并将结果按读取顺序存储在一个完全覆盖该对象的 `unsigned char` 数组中。流的文件位置指示器（如果已定义）应按成功读取的字节数前进。如果发生错误，流的文件位置指示器的结果值是未指定的。如果读取了部分元素，其值是未指定的。

`fread()` 函数可能会标记与 `stream` 关联的文件的最后数据访问时间戳以供更新。最后一次数据访问时间戳应通过第一次成功执行使用 `stream` 的 `fgetc()`、`fgets()`、`fread()`、`fscanf()`、`getc()`、`getchar()`、`getdelim()`、`getline()` 或 `scanf()` 函数来标记更新，这些函数返回的数据不是由先前对 `ungetc()` 的调用提供的。

RETURN VALUE

`fread()` 函数应返回成功读取的元素数量，该数量仅当遇到错误或文件结束条件或 `size` 为零时才小于 `nitems`。如果 `size` 或 `nitems` 为 0，`fread()` 应返回 0，且数组的内容和流的状态应保持不变。否则，如果发生错误，应设置流的错误指示器，并应设置 `errno` 以指示错误。

ERRORS

参考 `fgetc()`。

以下章节为参考信息。

EXAMPLES

从流中读取

以下示例将单个100字节定长记录从 `fp` 流传输到 `buf` 所指向的数组。

```
#include <stdio.h>
...
size_t elements_read;
char buf[100];
FILE *fp;
...
elements_read = fread(buf, sizeof(buf), 1, fp);
...
```

如果发生读取错误, `elements_read` 将为零, 但从流中读取的字节数可能从零到 `sizeof(buf)` -1 之间的任何值。

以下示例从 `fp` 流中读取多个单字节元素到 `buf` 所指向的数组。

```
#include <stdio.h>
...
size_t bytes_read;
char buf[100];
FILE *fp;
...
bytes_read = fread(buf, 1, sizeof(buf), fp);
...
```

如果发生读取错误, `bytes_read` 将包含从流中读取的字节数。

APPLICATION USAGE

必须使用 `ferror()` 或 `feof()` 函数来区分错误条件和文件结束条件。

由于元素长度和字节顺序可能存在差异, 使用 `fwrite()` 写入的文件是应用程序相关的, 可能无法被不同的应用程序或同一应用程序在不同处理器上使用 `fread()` 读取。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准I/O流

- `feof()`

- `ferror()`

- `fgetc()`

- `fopen()`

- `fscanf()`

- `getc()`

-

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

为了与 ISO/IEC 9899:1999 标准对齐，进行了以下更改：

- 更新了 `fread()` 原型。
- 更新了 DESCRIPTION 以描述如何存储来自 `fgetc()` 调用的字节。

Issue 7

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0178 [232] 和 XSH/TC1-2008/0179 [14]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0129 [926]。

Issue 8

应用了 Austin Group Defect 1196，阐明了 RETURN VALUE 部分。

应用了 Austin Group Defect 1330，删除了过时的接口。

应用了 Austin Group Defect 1624，更改了 RETURN VALUE 部分。

1.50. free — 释放已分配的内存

概要

```
#include <stdlib.h>

void free(void *ptr);
```

描述

`free()` 函数应导致由 `ptr` 指向的空间被释放；即，使其可用于进一步的分配。如果 `ptr` 是空指针，则不应发生任何操作。否则，如果参数不匹配先前由 `aligned_alloc()`、`calloc()`、`malloc()`、`posix_memalign()`、`realloc()`、`reallocarray()` 或 POSIX.1-2024 中如同 `malloc()` 一样分配内存的函数返回的指针，或者如果该空间已通过 `free()`、`reallocarray()` 或 `realloc()` 调用释放，则行为是未定义的。

对指向已释放空间的指针的任何使用都会导致未定义行为。

如果 `ptr` 是空指针或先前如同由 `malloc()` 返回且尚未释放的指针，`free()` 函数不应修改 `errno`。

为了确定数据竞争的存在性，`free()` 应表现得好像它只访问通过其参数可访问的内存位置，而不访问其他静态持续时间存储。但是，该函数可以可见地修改其释放的存储。分配或释放特定内存区域的 `aligned_alloc()`、`calloc()`、`free()`、`malloc()`、`posix_memalign()`、`reallocarray()` 和 `realloc()` 调用应发生在单一的总顺序中，并且每个此类释放调用应与此顺序中的下一个分配（如果有）同步。

返回值

`free()` 函数不应返回值。

错误

未定义任何错误。

示例

无。

应用程序使用

现在不再要求实现支持包含 `<malloc.h>`。

因为 `free()` 函数对于有效指针不会修改 `errno`，所以可以安全地在清理代码中使用它而不会破坏之前的错误，如以下示例代码：

```
// buf 是通过 malloc(buflen) 获得的
ret = write(fd, buf, buflen);
if (ret < 0) {
    free(buf);
    return ret;
}
```

但是，本标准的早期版本没有要求这一点，相同的示例必须写为：

```
// buf 是通过 malloc(buflen) 获得的
ret = write(fd, buf, buflen);
if (ret < 0) {
    int save = errno;
    free(buf);
    errno = save;
    return ret;
}
```

原理

无。

未来方向

无。

另请参阅

- `aligned_alloc()`

- `calloc()`
- `malloc()`
- `posix_memalign()`
- `realloc()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1

源自 SVID 的 Issue 1。

Issue 6

删除了对 `valloc()` 函数的引用。

Issue 7

更新了 DESCRIPTION，以阐明如果返回的指针不是由如同 `malloc()` 一样分配内存的函数返回的，则行为是未定义的。

Issue 8

应用了 Austin Group Defect 385，增加了 `free()` 在传递给可释放对象的指针时不修改 `errno` 的要求。

应用了 Austin Group Defect 1218，增加了 `reallocarray()`。

应用了 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准保持一致。

1.51. freopen

SYNOPSIS

```
#include <stdio.h>

FILE *freopen(const char *restrict pathname,
              const char *restrict mode,
              FILE *restrict stream);
```

DESCRIPTION

[Option Start] 除了"独占访问"要求（参见 `fopen()`）外，本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何其他冲突都是无意的。POSIX.1-2024 在除"独占访问"外的所有 `freopen()` 功能上遵循 ISO C 标准。[Option End]

`freopen()` 函数应首先尝试刷新与 `stream` 关联的流，如同调用 `fflush(stream)` 一样。未能成功刷新流应被忽略。如果 `pathname` 不是空指针，`freopen()` 应关闭与 `stream` 关联的任何文件描述符。未能成功关闭文件描述符应被忽略。流的错误和文件结束指示器应被清除。

`freopen()` 函数应打开路径名为 `pathname` 指向的字符串的文件，并将 `stream` 指向的流与其关联。`mode` 参数的使用方式应与 `fopen()` 中的相同。

无论后续打开是否成功，原始流都应被关闭。

如果 `pathname` 是空指针，`freopen()` 函数应尝试将流的模式更改为 `mode` 指定的模式，如同使用了当前与流关联的文件名一样。在这种情况下，如果对 `freopen()` 的调用成功，与流关联的文件描述符不需要被关闭。允许哪些模式更改（如果有）以及在什么情况下允许更改，是由实现定义的。

成功调用 `freopen()` 函数后，流的定向应被清除，[Option Start] 编码规则应被清除，[Option End] 并且关联的 `mbstate_t` 对象应被设置为描述初始转换状态。

[Option Start] 如果 `pathname` 不是空指针，或者 `pathname` 是空指针但指定的模式更改需要关闭并重新打开与流关联的文件描述符，则与重新打开的流关联的文件描述符应被分配并打开，如同通过使用为具有相同 `mode` 参数的 `fopen()` 指定的标志调用 `open()` 一样。[Option End]

RETURN VALUE

成功完成后，`freopen()` 应返回 `stream` 的值。否则，应返回空指针，[Option Start] 并设置 `errno` 以指示错误。[Option End]

ERRORS

`freopen()` 函数在以下情况下应失败：

- **[EACCES]**

[Option Start] 路径前缀的某个组件的搜索权限被拒绝，或者文件存在且 `mode` 指定的权限被拒绝，或者文件不存在且要创建的文件的父目录的写权限被拒绝。[Option End]

- **[EBADF]**

[Option Start] 当 `pathname` 是空指针时，流底层的基础文件描述符不是有效的文件描述符。[Option End]

- **[EILSEQ]**

[Option Start] `mode` 参数以 `w` 或 `a` 开始，文件之前不存在，并且最后一个路径名组件不是可移植的文件名，无法在目标目录中创建。[Option End]

- **[EEXIST]**

[Option Start] `mode` 参数以 `w` 或 `a` 开始并包含 `x`，但文件已存在。[Option End]

- **[EINTR]**

[Option Start] 在 `freopen()` 执行期间捕获到信号。[Option End]

- **[EISDIR]**

[Option Start] 命名的文件是目录，且 `mode` 要求写访问。[Option End]

- **[ELOOP]**

[Option Start] 在解析 `pathname` 参数期间遇到符号链接循环。[Option End]

- **[EMFILE]**

[Option Start] 进程可用的所有文件描述符当前都已打开。[Option End]

- **[ENAMETOOLONG]**

[Option Start] 路径名的某个组件的长度超过 `{NAME_MAX}`。[Option End]

- **[ENFILE]**

[Option Start] 系统中当前打开的文件数已达到最大允许数量。[Option End]

End]

- **[ENOENT]**

[Option Start] mode 字符串以 'r' 开始且 pathname 的某个组件没有命名现有文件，或者 mode 以 'w' 或 'a' 开始且 pathname 的路径前缀的某个组件没有命名现有文件，或者 pathname 是空字符串。[Option End]

- **[ENOENT] 或 [ENOTDIR]**

[Option Start] pathname 参数包含至少一个非 字符并以一个或多个尾随字符结尾。如果去掉尾随 字符的 pathname 会命名现有文件，则不应出现 [ENOENT] 错误。[Option End]

- **[ENOSPC]**

[Option Start] 包含新文件的目录或文件系统无法扩展，文件不存在，且本应创建该文件。[Option End]

- **[ENOTDIR]**

[Option Start] 路径前缀的某个组件命名了现有文件，该文件既不是目录也不是指向目录的符号链接，或者 pathname 参数包含至少一个非 字符并以一个或多个尾随 字符结尾，且最后一个路径名组件命名了现有文件，该文件既不是目录也不是指向目录的符号链接。[Option End]

- **[ENXIO]**

[Option Start] 命名的文件是字符特殊文件或块特殊文件，并且与此特殊文件关联的设备不存在。[Option End]

- **[EOVERFLOW]**

[Option Start] 命名的文件是常规文件，且文件大小无法在 off_t 类型的对象中正确表示。[Option End]

- **[EROFS]**

[Option Start] 命名的文件位于只读文件系统上，且 mode 要求写访问。[Option End]

freopen() 函数在以下情况下可能失败：

- **[EBADF]**

[Option Start] 当 pathname 是空指针时，打开流底层文件描述符的模式不支持请求的模式。[Option End]

- **[EINVAL]**

[Option Start] mode 参数的值无效。[Option End]

- **[ELOOP]**

[Option Start] 在解析 pathname 参数期间遇到超过 {SYMLOOP_MAX} 个符号链接。[Option End]

- **[ENAMETOOLONG]**

[Option Start] 路径名的长度超过 {PATH_MAX}，或者符号链接的路径名解析产生的中间结果长度超过 {PATH_MAX}。[Option End]

- **[ENOMEM]**

[Option Start] 可用存储空间不足。[Option End]

- **[ENXIO]**

[Option Start] 对不存在的设备发出请求，或者请求超出了设备的功能范围。[Option End]

- **[ETXTBSY]**

[Option Start] 文件是正在执行的纯过程（共享文本）文件，且 mode 要求写访问。[Option End]

以下各节是提供信息的。

EXAMPLES

将标准输出重定向到文件

以下示例将所有标准输出记录到 /tmp/logfile 文件中。

```
#include <stdio.h>
...
FILE *fp;
...
fp = freopen ("/tmp/logfile", "a+", stdout);
...
```

APPLICATION USAGE

freopen() 函数通常用于将与 stdin、stdout 和 stderr 关联的预打开流附加到其他文件。

由于实现不要求在 pathname 参数为 NULL 时支持任何流模式更改，可移植的应用程序不能依赖使用 freopen() 来更改流模式，不鼓励使用此功能。此功能最初被添加到 ISO C 标准是为了便于将 stdin 和 stdout 更改为二进制模式。由于 mode 中的 'b' 字符在 POSIX 系统上没有效果，在 POSIX 应用程序中不需要使用此功能。但是，即使 'b' 被忽略，成功调用 freopen(NULL, "wb", stdout) 确

实有效果。特别是，对于常规文件，它会截断文件并将流的文件位置指示器设置为文件的开头。这些副作用可能是 ISO/IEC 9899:1999 标准（以及当前标准）中指定此功能的方式的意外后果，但除非或直到 ISO C 标准被更改，成功调用 `freopen(NULL, "wb", stdout)` 的应用程序在符合标准的系统上的情况下会以意想不到的方式表现，例如：

```
{ appl file1; appl file2; } > file3
```

这将导致 `file3` 仅包含第二次调用 `appl` 的输出。

另请参见 `fopen()` 的 APPLICATION USAGE。

RATIONALE

参见 `fopen()` 的 RATIONALE。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准 I/O 流
- `fclose()`
- `fdopen()`
- `fflush()`
- `fmemopen()`
- `fopen()`
- `mbsinit()`
- `open()`
- `open_memstream()`
-

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

DESCRIPTION 更新为指示流的定向在成功调用 `freopen()` 函数后被清除，流的转换状态被设置为初始转换状态。

添加了大文件峰会扩展。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源自与单一 UNIX 规范的对齐：

- 在 DESCRIPTION 中，添加了文本以指示在打开文件描述中设置偏移量最大值。此更改是为了支持大文件。
- 在 ERRORS 部分中，添加了 [EOVERFLOW] 条件。此更改是为了支持大文件。
- 添加了 [ELOOP] 强制错误条件。
- 添加了第二个 [ENAMETOOLONG] 作为可选错误条件。
- 添加了 [EINVAL]、[ENOMEM]、[ENXIO] 和 [ETXTBSY] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐进行了以下更改：

- 更新了 `freopen()` 原型。
- 更新了 DESCRIPTION。

强制 [ELOOP] 错误条件的措辞已更新，并添加了第二个可选 [ELOOP] 错误条件。

关于关闭失败的 DESCRIPTION 已更新，将"文件"更改为"文件描述符"。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/40，在 DESCRIPTION 中添加以下句子："在这种情况下，如果对 `freopen()` 的调用成功，与流关联的文件描述符不需要被关闭。"。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/41，向 ERRORS 部分添加了强制 [EBADF] 错误和可选 [EBADF] 错误。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #043，阐明 `freopen()` 函数按照 `open()` 的方式分配文件描述符。

应用 Austin Group Interpretation 1003.1-2001 #143。

应用 Austin Group Interpretation 1003.1-2001 #159，阐明对打开文件描述上设置的标志的要求。

应用 SD5-XBD-ERN-4，更改 [EMFILE] 错误的定义。

应用 SD5-XSH-ERN-150 和 SD5-XSH-ERN-219。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0181 [291,433], XSH/TC1-2008/0182 [146,433], XSH/TC1-2008/0183 [324]，和 XSH/TC1-2008/0184 [14]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0134 [822]。

Issue 8

应用 Austin Group Defect 293，添加 [EILSEQ] 错误。

应用 Austin Group Defect 411，添加 e 和 x 模式字符串字符。

应用 Austin Group Defect 1200，更正 [ELOOP] 错误中的参数名称。

应用 Austin Group Defect 1302，将此函数与 ISO/IEC 9899:2018 标准对齐。

信息性文本结束。

1.52. fscanf

概要

```
#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format,
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format,
```

描述

[CX] 本参考页描述的功能与ISO C标准一致。此处描述的要求与ISO C标准之间的任何冲突都是非故意的。本POSIX.1-2024卷遵循ISO C标准。

`fscanf()` 函数应从指定的输入流读取。 `scanf()` 函数应从标准输入流stdin读取。 `sscanf()` 函数应从字符串 `s` 读取。每个函数读取字节，根据格式进行解释，并将结果存储在其参数中。每个函数期望作为参数的是一个如下所述的控制字符串格式，以及一组指示转换后的输入应存储在何处的指针参数。如果格式参数不足，结果是未定义的。如果格式已耗尽但仍有参数剩余，多余参数应被求值但否则被忽略。

[CX] 转换可以应用于参数列表中格式之后的第n个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符%（见下文）被序列"%n\$"替换，其中n是范围[1,{NL_ARGMAX}]内的十进制整数。此功能提供了定义按适合特定语言的顺序选择参数的格式字符串的方法。在包含"%n\$"形式转换说明的格式字符串中，格式字符串是否可以多次引用参数列表中的编号参数是未指定的。

格式可以包含两种形式的转换说明符，即%或"%n\$"，但两种形式不能在单个格式字符串中混合使用。唯一的例外是%%或%*可以与"%n\$"形式混合使用。当使用编号参数规范时，指定第N个参数要求所有前导参数，从第一个到第(N-1)个，都是指针。

所有形式的 `fscanf()` 函数都应允许在输入字符串中检测依赖于语言的小数点字符。小数点字符在当前语言环境（类别LC_NUMERIC）中定义。在POSIX语言环境中，或在未定义小数点字符的语言环境中，小数点字符应默认为句点('.')。

格式字符串

应用程序应确保格式是字符字符串，如果存在，则以其初始移位状态开始和结束，由零个或多个指令组成。每个指令由以下内容之一组成：一个或多个空白字节；一个普通字符（既不是 '%' 也不是空白字节）；或一个转换说明。每个转换说明由字符 '%' [CX] 或字符序列 "%n\$" 引入，之后按顺序出现以下内容：

- 可选的赋值抑制字符 '*'。
- 可选的非零十进制整数，指定最大字段宽度。
- [CX] 可选的赋值分配字符 'm'。
- 可选的长度修饰符，指定接收对象的大小。
- 转换说明符字符，指定要应用的转换类型。

`fscanf()` 函数应依次执行格式的每个指令。当所有指令都已执行，或者如果指令失败（如下详述），函数应返回。失败被描述为输入失败（由于输入字节不可用）或匹配失败（由于不适当的输入）。

指令执行

由一个或多个空白字节组成的指令应通过读取输入直到第一个非空白字节来执行，该字节应保持未读状态，或者直到无法读取更多字节。该指令永远不会失败。

作为普通字符的指令应按以下方式执行：从输入中读取下一个字节，并与组成指令的字节进行比较；如果比较显示它们不等价，指令应失败，不同的和后续的字节应保持未读状态。类似地，如果文件结束、编码错误或读取错误阻止读取字符，指令应失败。

作为转换说明的指令定义了一组匹配输入序列，如下每个转换字符所述。转换说明应按以下步骤执行：

1. 应跳过输入空白字节，除非转换说明包含 [、c、C 或 n 转换说明符。
2. 应从输入中读取一个项目，除非转换说明包含 n 转换说明符。输入项目应定义为最长序列的输入字节（达到任何指定的最大字段宽度，该宽度可能根据转换说明符以字符或字节测量），它是匹配序列的初始子序列。输入项目之后的第一个字节（如果有）应保持未读状态。如果输入项目的长度为 0，转换说明的执行应失败；这种情况是匹配失败，除非文件结束、编码错误或读取错误阻止从流读取输入，在这种情况下是输入失败。
3. 除了 % 转换说明符的情况外，输入项目（或者在 %n 转换说明符的情况下，输入字节的计数）应被转换为适合转换字符的类型。如果输入项目不是匹配序列，转换说明的执行失败；这种情况是匹配失败。除非赋值抑制

由'*'指示，转换的结果应放置在格式参数之后尚未接收转换结果的第一个参数指向的对象中，如果转换说明由%引入，[CX]或者由字符序列"%n\$"引入，则放在第n个参数中。如果此对象没有适当的类型，或者转换的结果无法在提供的空间中表示，则行为是未定义的。

[CX] c、s和[转换说明符应接受可选的赋值分配字符'm'，这将导致分配内存缓冲区来保存转换结果。如果转换说明符是s或[，分配的缓冲区应包括用于终止空字符串（或宽字符）的空间。在这种情况下，对应于转换说明符的参数应该是指针变量的引用，该变量将接收指向分配缓冲区的指针。系统应分配缓冲区，就像调用了 `malloc()` 一样。应用程序应负责在使用后释放内存。如果没有足够的内存分配缓冲区，函数应将errno设置为[ENOMEM]并导致转换错误。如果函数返回EOF，此调用中使用赋值分配字符'm'为参数成功分配的任何内存应在函数返回前被释放。

长度修饰符

长度修饰符及其含义是：

hh

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `signed char` 或 `unsigned char` 指针的参数。

h

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `short` 或 `unsigned short` 指针的参数。

l (ell)

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `long` 或 `unsigned long` 指针的参数；后续的a、A、e、E、f、F、g或G转换说明符应用于类型为 `double` 指针的参数；或者后续的c、s或[转换说明符应用于类型为 `wchar_t` 指针的参数。[CX]如果指定了'm'赋值分配字符，转换应用于类型为指向 `wchar_t` 指针的参数。

ll (ell-ell)

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `long long` 或 `unsigned long long` 指针的参数。

j

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `intmax_t` 或 `uintmax_t` 指针的参数。

z

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `size_t` 或相应有符号整数类型的指针的参数。

t

指定后续的d、i、o、u、x、X或n转换说明符应用于类型为 `ptrdiff_t` 或相应的 `unsigned` 类型指针的参数。

L

指定后续的a、A、e、E、f、F、g或G转换说明符应用于类型为 `long double` 指针的参数。

如果长度修饰符与除上述指定之外的任何转换说明符一起出现，行为是未定义的。

转换说明符

以下转换说明符是有效的：

d

匹配可选有符号十进制整数，其格式与 `strtol()` 的主体序列期望的格式相同，base参数值为10。在没有大小修饰符的情况下，应用程序应确保相应参数是 `int` 指针。

i

匹配可选有符号整数，其格式与 `strtol()` 的主体序列期望的格式相同，base参数值为0。在没有大小修饰符的情况下，应用程序应确保相应参数是 `int` 指针。

o

匹配可选有符号八进制整数，其格式与 `strtoul()` 的主体序列期望的格式相同，base参数值为8。在没有大小修饰符的情况下，应用程序应确保相应参数是 `unsigned` 指针。

u

匹配可选有符号十进制整数，其格式与 `strtoul()` 的主体序列期望的格式相同，base参数值为10。在没有大小修饰符的情况下，应用程序应确保相应参数是 `unsigned` 指针。

x

匹配可选有符号十六进制整数，其格式与 `strtoul()` 的主体序列期望的格式相同，base参数值为16。在没有大小修饰符的情况下，应用程序应确保相应参数是 `unsigned` 指针。

a, e, f, g

匹配可选有符号浮点数、无穷大或NaN，其格式与 `strtod()` 的主体序列期望的格式相同。在没有大小修饰符的情况下，应用程序应确保相应参数是 `float` 指针。

如果 `fprintf()` 函数族为无穷大和NaN（以浮点格式编码的符号实体）生成字符串表示以支持IEEE Std 754-1985，`fscanf()` 函数族应将它们识别为输入。

s

匹配非空白字节的字节序列。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列和自动添加的终止空字符代码。[CX]否则，应用程序应确保相应参数是指向 `char` 指针的指针。

如果存在l (ell) 限定符，输入是以初始移位状态开始的字符序列。每个字符应转换为宽字符，就像通过调用 `mbrtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受序列和自动添加的终止空宽字符。[CX]否则，应用程序应确保相应参数是指向 `wchar_t` 指针的指针。

[

从一组期望字节（扫描集）中匹配非空字节序列。在这种情况下，正常跳过空白字节应被抑制。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列和自动添加的终止空字节。[CX]否则，应用程序应确保相应参数是指向 `char` 指针的指针。

如果存在l (ell) 限定符，输入是以初始移位状态开始的字符序列。序列中的每个字符应转换为宽字符，就像通过调用 `mbrtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受序列和自动添加的终止空宽字符。[CX]否则，应用程序应确保相应参数是指向 `wchar_t` 指针的指针。

转换说明包括格式字符串中所有后续字节，直到并包括匹配的右方括号(']')。方括号之间的字节（扫描列表）组成扫描集，除非左方括号之后的字节是抑扬符('^')，在这种情况下，扫描集包含不出现在抑扬符和右方括号之间扫描列表中的所有字节。如果转换说明以"[]"或"[^]"开始，右方括号包含在扫描列表中，下一个右方括号是结束转换说明的匹配右方括号；否则，第一个右方括号是结束转换说明的那个。如果'-'在扫描列表中，并且不是第一个字符，也不是第二个字符（其中第一个字符是'^'），也不是最后一个字符，行为是实现定义的。

c

匹配由字段宽度指定的数字的字节序列（如果转换说明中不存在字段宽度则为1）。不添加空字节。在这种情况下，正常跳过空白字节应被抑制。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列。[CX]否则，应用程序应确保相应参数是指向 `char` 指针的指针。

如果存在l (ell) 限定符，输入应该是以初始移位状态开始的字符序列。序列中的每个字符转换为宽字符，就像通过调用 `mbtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。不添加空宽字符。如果未指定'm'赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受生成的宽字符序列。[CX]否则，应用程序应确保相应参数是指向 `wchar_t` 指针的指针。

p

匹配实现定义的序列集合，该集合应与相应 `fprintf()` 函数的%p转换说明符产生的序列集合相同。应用程序应确保相应参数是指向 `void` 指针的指针。输入项的解释是实现定义的。如果输入项是同一程序执行期间早期转换的值，结果指针应与该值相等；否则，%p转换说明符的行为是未定义的。

n

不消耗输入。应用程序应确保相应参数是指向整数的指针，该整数将被写入此 `fscanf()` 函数调用迄今为止从输入中读取的字节数。执行%on转换说明符不应增加函数执行完成时返回的赋值计数。不应转换任何参数，但应消耗一个参数。

%

匹配单个'%'字符；不应进行赋值或转换。完整的转换说明应为%%。

返回值

成功完成时，这些函数应返回成功匹配和赋值的输入项数量；在早期匹配失败的情况下，此数量可以为零。如果输入在第一次匹配失败或转换之前结束，应返回EOF。如果发生读取错误，流的错误指示符应被设置，应返回EOF，并应设置 `errno` 以指示错误。

错误

有关函数应失败和可能失败的条件，请参阅底层 `fgetc()` 或 `fgetwc()` 函数的文档。此外，`fscanf()` 函数可能在以下情况下失败：

- **[ENOMEM]** - 使用'm'赋值分配字符时没有足够的存储空间来保存转换后的值。

示例

```
#include <stdio.h>

int main(void) {
    int i;
    float f;
    char s[100];

    /* 读取整数、浮点数和字符串 */
    if (fscanf(stdin, "%d %f %99s", &i, &f, s) == 3) {
        printf("Read: %d, %f, %s\n", i, f, s);
    }

    return 0;
}
```

应用程序使用

fscanf() 函数族可用于需要从文件、标准输入或字符串读取和解析格式化输入的应用程序中。格式字符串提供了对输入解释和转换的灵活控制。

赋值分配字符'm'在输入大小事先未知时特别有用，因为它允许实现自动分配适当的内存。

原理

fscanf() 函数为输入解析和转换提供了强大的机制。该设计允许：

- 通过格式规范对输入解析进行精确控制
- 使用'm'修饰符时为可变长度输入自动分配内存
- 支持窄字符和宽字符输入
- 为国际化应用程序提供编号参数规范

行为被仔细指定以确保在不同实现上获得一致的结果，同时保持与ISO C标准的兼容性。

未来方向

无。

另请参见

[fgetc\(\)](#) , [fgetwc\(\)](#) , [fprintf\(\)](#) , [fread\(\)](#) , [fseek\(\)](#) , [getchar\(\)](#) ,
[scanf\(\)](#) , [setlocale\(\)](#) , [sscanf\(\)](#) , [strtod\(\)](#) , [strtol\(\)](#) ,
[strtoul\(\)](#) , [ungetc\(\)](#)

1.53. fsync

SYNOPSIS

```
#include <unistd.h>

int fsync(int fildes);
```

DESCRIPTION

`fsync()` 函数应当请求将 `fildes` 指定的打开文件描述符的所有数据传输到与该文件关联的存储设备。传输的性质由实现定义。`fsync()` 函数在系统完成该操作或检测到错误之前不应返回。

如果定义了 `_POSIX_SYNCHRONIZED_IO`，`fsync()` 函数应强制将与文件描述符 `fildes` 关联的所有当前排队的 I/O 操作转为同步 I/O 完成状态。所有 I/O 操作应按照同步 I/O 文件完整性完成的规定完成。

RETURN VALUE

成功完成后，`fsync()` 应返回 0。否则，应返回 -1 并设置 `errno` 以指示错误。如果 `fsync()` 函数失败，不保证未完成的 I/O 操作已完成。

ERRORS

`fsync()` 函数在以下情况下应失败：

- **[EBADF]**
`fildes` 参数不是有效的描述符。
- **[EINTR]**
`fsync()` 函数被信号中断。
- **[EINVAL]**
`fildes` 参数不指向支持此操作的文件。
- **[EIO]**
在从文件系统读取或写入时发生 I/O 错误。

如果任何排队的 I/O 操作失败，`fsync()` 应返回为 `read()` 和 `write()` 定义的错误条件。

以下部分为补充信息。

EXAMPLES

无。

APPLICATION USAGE

需要在继续执行前完成文件修改的程序应使用 `fsync()` 函数；例如，包含简单事务功能的程序可能使用它来确保事务引起的所有对文件的修改都被记录。

修改目录的应用程序（例如在目录中创建文件）可以对目录调用 `fsync()` 以确保目录条目和文件属性同步。对于大多数应用程序，不需要同步目录条目（参见 XBD 4.11 文件系统缓存）。

RATIONALE

`fsync()` 函数旨在强制将数据从缓冲区缓存进行物理写入，并确保在系统崩溃或其他故障之后，截至 `fsync()` 调用时的所有数据都记录在磁盘上。由于“缓冲区缓存”、“系统崩溃”、“物理写入”和“非易失性存储”的概念在此处未定义，措辞必须更加抽象。

如果未定义 `_POSIX_SYNCHRONIZED_IO`，措辞在很大程度上依赖一致性文档来告知用户可以从系统预期什么。明确允许空实现。这在系统在任何情况下都无法保证非易失性存储，或者系统具有高度容错性且不需要此功能的情况下可能是有效的。在这些极端情况之间的中间地带，`fsync()` 可能或可能不会实际导致数据写入到能够抵御电源故障的位置。一致性文档应至少标识存在一种配置（以及如何获得该配置），其中至少可以确保用户可以选择用于关键数据的某些文件的这种安全。不要求提供详尽列表，而是提供足够的信息，以便在需要保存关键数据时，用户可以确定如何配置系统以允许数据写入非易失性存储。

可以合理地断言，`fsync()` 的关键方面在测试套件中测试是不合理的。这并不会使函数价值降低，只是使其更难测试。正式的一致性测试可能应该在此条件的测试期间强制系统崩溃（电源关闭），但需要以这种方式进行，使得自动化测试除在制作正式结果记录时之外不需要这样做。省略对 `fsync()` 的测试也是合理的，允许将其视为实现质量问题。

FUTURE DIRECTIONS

无。

SEE ALSO

[sync\(\)](#)

XBD [<unistd.h>](#)

CHANGE HISTORY

首次发布于 Issue 3。

Issue 5

与 POSIX 实时扩展中的 [fsync\(\)](#) 对齐。具体来说，DESCRIPTION 和 RETURN VALUE 部分大幅扩展，ERRORS 部分更新为指示 [fsync\(\)](#) 可以返回为 [read\(\)](#) 和 [write\(\)](#) 定义的错误条件。

Issue 6

此函数被标记为文件同步选项的一部分。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 添加了 [\[EINVAL\]](#) 和 [\[EIO\]](#) 强制错误条件。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/44，对 DESCRIPTION 进行了编辑性重新措辞。不打算改变含义。

Issue 8

应用了 Austin Group 缺陷 672，更改了 APPLICATION USAGE 部分。

补充信息结束。

1.54. ftrylockfile

SYNOPSIS

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

DESCRIPTION

这些函数应提供对与标准 I/O 流 (standard I/O streams) 相关联的锁进行显式应用程序级锁定 (参见 [2.5 标准输入输出流](#))。线程可以使用这些函数来划分作为一个单元执行的一系列 I/O 语句。

`flockfile()` 函数应为线程获取 (`FILE *`) 对象的所有权。

`ftrylockfile()` 函数应在对象可用时为线程获取 (`FILE *`) 对象的所有权；

`ftrylockfile()` 是 `flockfile()` 的非阻塞版本。

`funlockfile()` 函数应放弃授予线程的所有权。如果除当前所有者之外的线程调用 `funlockfile()` 函数，则行为未定义。

这些函数的行为应如同每个 (`FILE *`) 对象都有一个关联的锁计数器。此计数器在创建 (`FILE *`) 对象时隐式初始化为零。当计数器为零时, (`FILE *`) 对象处于未锁定状态。当计数器为正数时, 单个线程拥有该 (`FILE *`) 对象。当调用 `flockfile()` 函数时, 如果计数器为零, 或者计数器为正数且调用者拥有该 (`FILE *`) 对象, 则计数器应递增。否则, 调用线程应被挂起, 等待计数器返回零。每次调用 `funlockfile()` 都会使计数器递减。这允许匹配的 `flockfile()` (或成功的 `ftrylockfile()`) 和 `funlockfile()` 调用嵌套。

RETURN VALUE

`flockfile()` 和 `funlockfile()` 无返回值。

`ftrylockfile()` 函数成功时应返回零, 无法获取锁时应返回非零值。

ERRORS

未定义错误。

以下部分为提供信息的部分。

EXAMPLES

无。

APPLICATION USAGE

使用这些函数的应用程序可能会受到优先级反转 (priority inversion) 的影响，如 XBD 3.275 优先级反转 中所述。

对 `_exit()` 的调用可能会阻塞，直到锁定的流被解锁，因为拥有 (`FILE *`) 对象所有权的线程会阻止其他线程访问该 (`FILE *`) 对象的所有函数调用 (名称以 `_unlocked` 结尾的函数除外)，包括对 `_exit()` 的调用。

注意：`FILE` 锁不是文件锁 (参见 XBD 3.143 文件锁)。

RATIONALE

`flockfile()` 和 `funlockfile()` 函数为每个 `FILE` 提供一个正交的互斥锁。`ftrylockfile()` 函数提供获取 `FILE` 锁的非阻塞尝试，类似于 `pthread_mutex_trylock()`。

这些锁的行为应与 `stdio` 内部用于线程安全的锁相同。这既提供了这些函数的线程安全性，而不需要第二级的内部锁定，也允许 `stdio` 中的函数以其他 `stdio` 函数为基础实现。

应用程序开发者和实现者应该意识到 `FILE` 对象存在潜在的死锁问题。例如，`stdio` 的行缓冲刷新语义 (通过 `{_IOLBF}` 请求) 要求某些输入操作有时会刷新实现定义的行缓冲输出流的缓冲内容。如果两个线程各自持有对方 `FILE` 的锁，就会产生死锁。这种类型的死锁可以通过以一致的顺序获取 `FILE` 锁来避免。特别是，如果一个线程需要同时获取输入流和输出流的锁，通常可以通过先获取输入流的锁再获取输出流的锁来避免行缓冲输出流的死锁。

总之，与其他线程共享 `stdio` 流的线程可以使用 `flockfile()` 和 `funlockfile()` 来保持单个线程执行的 I/O 序列捆绑在一起。唯一需要使用 `flockfile()` 和 `funlockfile()` 的情况是为使用 `*_unlocked` 函数/宏提供保护作用域。这将成本/性能权衡移动到最优点。

FUTURE DIRECTIONS

无。

SEE ALSO

`_exit()`, `getc_unlocked()`

XBD 3.275 优先级反转, `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 5

包含用于与 POSIX 线程扩展对齐。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

Issue 7

`flockfile()`、`ftrylockfile()` 和 `funlockfile()` 函数从线程安全函数选项移动到基础规范。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0140 [118]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0116 [611]。

Issue 8

应用了 Austin Group Defect 1118, 澄清 `FILE` 锁不是文件锁。

应用了 Austin Group Defect 1302, 将部分文本替换为对 2.5 标准输入输出流 的引用。

信息文本结束。

1.55. flockfile, ftrylockfile, funlockfile — stdio 锁定函数

概要

```
#include <stdio.h>

void flockfile(FILE *file);
int ftrylockfile(FILE *file);
void funlockfile(FILE *file);
```

描述

这些函数应提供与标准 I/O 流相关联的锁的显式应用级锁定（参见 [2.5 Standard I/O Streams](#)）。线程可以使用这些函数来界定作为单元执行的一系列 I/O 语句。

`flockfile()` 函数应为线程获取 (`FILE *`) 对象的所有权。

`ftrylockfile()` 函数在对象可用时为线程获取 (`FILE *`) 对象的所有权；
`ftrylockfile()` 是 `flockfile()` 的非阻塞版本。

`funlockfile()` 函数应放弃授予线程的所有权。如果当前所有者之外的线程调用 `funlockfile()` 函数，则行为未定义。

这些函数的行为应表现为每个 (`FILE *`) 对象都有一个与之关联的锁计数。此计数在创建 (`FILE *`) 对象时隐式初始化为零。当计数为零时，(`FILE *`) 对象处于未锁定状态。当计数为正时，单个线程拥有 (`FILE *`) 对象。当调用 `flockfile()` 函数时，如果计数为零，或者计数为正且调用者拥有 (`FILE *`) 对象，则计数应递增。否则，调用线程应被挂起，等待计数返回零。每次调用 `funlockfile()` 都会递减计数。这允许匹配的 `flockfile()` 调用（或成功的 `ftrylockfile()` 调用）和 `funlockfile()` 调用可以嵌套。

返回值

`flockfile()` 和 `funlockfile()` 无返回值。

`ftrylockfile()` 函数成功时应返回零，非零值表示无法获取锁。

错误

未定义任何错误。

以下各节为参考性内容。

示例

无。

应用用法

使用这些函数的应用程序可能会受到优先级反转的影响，如 XBD [3.275 Priority Inversion](#) 中所述。

对 `exit()` 的调用可能会阻塞，直到锁定的流被解锁，因为拥有 (`FILE *`) 对象所有权的线程会阻止其他线程中所有引用该 (`FILE *`) 对象的函数调用（名称以 `_unlocked` 结尾的函数除外），包括对 `exit()` 的调用。

注意： `FILE` 锁不是文件锁（参见 XBD [3.143 File Lock](#)）。

基本原理

`flockfile()` 和 `funlockfile()` 函数为每个 `FILE` 提供一个正交的互斥锁。`ftrylockfile()` 函数提供获取 `FILE` 锁的非阻塞尝试，类似于 `pthread_mutex_trylock()`。

这些锁的行为应表现为与 `stdio` 内部用于线程安全的锁相同。这既提供了这些函数的线程安全性，而无需第二级内部锁定，又允许 `stdio` 中的函数用其他 `stdio` 函数来实现。

应用程序开发者和实现者应该意识到 `FILE` 对象存在潜在的死锁问题。例如，`stdio` 的行缓冲刷新语义（通过 `{_IOLBF}` 请求）要求某些输入操作有时会导致实现定义的行缓冲输出流的缓冲内容被刷新。如果两个线程各自持有对方 `FILE` 的锁，就会发生死锁。这种类型的死锁可以通过以一致的顺序获取 `FILE` 锁来避免。特别是，如果线程既要获取输入流的锁又要获取输出流的锁，通常可以通过先获取输入流锁再获取输出流锁来避免行缓冲输出流的死锁。

总之，与其他线程共享 `stdio` 流的线程可以使用 `flockfile()` 和 `funlockfile()` 来使单个线程执行的 I/O 序列保持捆绑。唯一需要使用

`flockfile()` 和 `funlockfile()` 的情况是提供作用域保护 `*_unlocked` 函数/宏的使用。这将成本/性能权衡移至最优点。

未来方向

无。

另请参见

`exit()` , `getc_unlocked()`

XBD 3.275 Priority Inversion, `<stdio.h>`

变更历史

首次发布于 Issue 5

包含此内容以与 POSIX 线程扩展对齐。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

Issue 7

`flockfile()`、`ftrylockfile()` 和 `funlockfile()` 函数从线程安全函数选项移至基础规范。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0140 [118]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0116 [611]。

Issue 8

应用 Austin Group Defect 1118, 阐明 `FILE` 锁不是文件锁。

应用 Austin Group Defect 1302, 将部分文本替换为对 [2.5 Standard I/O Streams](#) 的引用。

参考性文本结束。

1.56. fwrite — 二进制输出

概要

```
#include <stdio.h>

size_t fwrite(const void *restrict ptr,
              size_t size,
              size_t nitems,
              FILE *restrict stream);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`fwrite()` 函数应从 `ptr` 指向的数组中，向 `stream` 指向的流写入最多 `nitems` 个元素，每个元素的大小由 `size` 指定。对于每个对象，应调用 `size` 次 `fputc()` 函数，从精确覆盖该对象的 `unsigned char` 数组中按顺序取值。流的文件位置指示器（如果已定义）应按成功写入的字节数前进。如果发生错误，流的文件位置指示器的最终值是未指定的。

[CX] 文件的数据最后修改时间戳和文件状态最后更改时间戳应被标记为更新，更新时机在 `fwrite()` 成功执行与下一次对同一流成功完成 `fflush()` 或 `fclose()` 调用之间，或调用 `exit()` 或 `abort()` 之间。

返回值

`fwrite()` 函数应返回成功写入的元素数量，只有在遇到写入错误时该数量才可能小于 `nitems`。如果 `size` 或 `nitems` 为 0，`fwrite()` 应返回 0 且流的状态保持不变。否则，如果发生写入错误，应设置流的错误指示器，[CX] 并且应设置 `errno` 来指示错误。

错误

参考 `fputc()`。

以下部分为参考信息。

示例

无。

应用用法

由于元素长度和字节顺序可能存在差异，使用 `fwrite()` 写入的文件是应用程序相关的，可能无法被不同应用程序或同一应用程序在不同处理器上使用 `fclose()` 读取。

原理

无。

未来方向

无。

参见

- 2.5 标准 I/O 流
- `ferror()`
- `fopen()`
- `fprintf()`
- `putc()`
- `puts()`
- `write()`
- XBD `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超越 ISO C 标准的扩展。

为与 ISO/IEC 9899:1999 标准对齐，进行了以下更改：

- 更新了 `fwrite()` 原型。
- 更新了描述部分，阐明如何使用 `fputc()` 写出数据。

Issue 7

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008 Technical Corrigendum 1, XSH/TC1-2008/0228 [14]。

Issue 8

应用了 Austin Group Defect 1196，阐明了返回值部分。

1.57. getc — 从流中获取一个字节

概要 (SYNOPSIS)

```
#include <stdio.h>

int getc(FILE *stream);
```

描述 (DESCRIPTION)

`getc()` 函数应等价于 `fgetc()`，但区别在于：如果它被实现为宏 (macro)，可能会对 *stream* 参数进行多次求值，因此该参数永远不应是具有副作用的表达式。

返回值 (RETURN VALUE)

参考 `fgetc()`。

错误 (ERRORS)

参考 `fgetc()`。

以下为补充信息章节。

示例 (EXAMPLES)

无。

应用用法 (APPLICATION USAGE)

如果 `getc()` 返回的整数值被存储到 `char` 类型的变量中，然后再与整数常量 EOF 进行比较，该比较可能永远不会成功，因为 `char` 类型变量在扩展为整数时的符号扩展是由实现定义的 (implementation-defined)。

由于 `getc()` 可能被实现为宏，它可能错误处理带有副作用的 *stream* 参数。特别是，`getc(*f_++)` 不一定能按预期工作。因此，在这种情况下，使用该函数之前应该先执行 "#undef getc"；也可以使用 `fgetc()` 代替。

基本原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- 2.5 标准输入输出流 (Standard I/O Streams)
- `fgetc()`
- XBD `<stdio.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008 技术勘误 1，XSH/TC1-2008/0231 [14]。

1.58. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — 带有显式客户 端锁定的 stdio 函数

概要 (SYNOPSIS)

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

描述 (DESCRIPTION)

应提供 `getc()`、`getchar()`、`putc()` 和 `putchar()` 函数的版本，分别命名为 `getc_unlocked()`、`getchar_unlocked()`、`putc_unlocked()` 和 `putchar_unlocked()`，这些版本在功能上等同于原始版本，但它们不需要以完全线程安全的方式实现。当在由 `flockfile()` (或 `ftrylockfile()`) 和 `funlockfile()` 保护的范围内使用时，它们应该是线程安全的。这些函数可以安全地用于多线程程序中，当且仅当调用线程拥有 `(FILE *)` 对象时才调用它们，这是在成功调用 `flockfile()` 或 `ftrylockfile()` 函数之后的情况。

如果 `getc_unlocked()` 或 `putc_unlocked()` 作为宏实现，它们可能会多次求值 `stream`，因此 `stream` 参数永远不应是有副作用的表达式。

返回值 (RETURN VALUE)

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

错误 (ERRORS)

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

示例 (EXAMPLES)

无。

应用程序使用 (APPLICATION USAGE)

由于它们可能作为宏实现，`getc_unlocked()` 和 `putc_unlocked()` 可能错误地处理有副作用的 `stream` 参数。特别是，`getc_unlocked(*f++)` 和 `putc_unlocked(c,*f++)` 不一定按预期工作。因此，在这种情况下使用这些函数之前应适当地添加以下语句：

```
#undef getc_unlocked
#undef putc_unlocked
```

原理说明 (RATIONALE)

出于性能原因，一些 I/O 函数通常作为宏实现（例如 `putc()` 和 `getc()`）。为了安全，它们需要同步，但在每个字符上进行同步通常开销太大。尽管如此，人们认为安全考虑更为重要；因此，`getc()`、`getchar()`、`putc()` 和 `putchar()` 函数被要求为线程安全的。然而，也提供了解锁版本，其名称清楚地表明了操作的不安全性质，但可以用来利用其更高的性能。这些解锁版本只能在使用导出的锁定原语显式锁定的程序区域内安全使用。特别是，像这样的序列：

```
lockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
unlockfile(fileptr);
```

是允许的，并且导致文本序列：

```
1
Line 2
```

被打印而不与其他线程的输出交错。

让标准名称如 `getc()`、`putc()` 等映射到“更快但不安全”而不是“更慢但安全”的版本是错误的。在任何一种情况下，当转换现有代码时，您仍然希望手动检查 `getc()`、`putc()` 等的所有使用。选择安全绑定作为默认值至少会产生正确的代码并保持“函数级别的原子性”不变量。否则会在转换后的代码中引入不

必要的同步错误。修改 `stdio (FILE *)` 结构或缓冲区的其他例程也被安全地同步。

注意，不需要像 `getc_locked()`、`putc_locked()` 等形式的函数，因为这是 `getc()`、`putc()` 等的功能。使用功能测试宏在 `getc_locked()` 和 `getc_unlocked()` 之间切换 `getc()` 的宏定义是不合适的，因为 ISO C 标准要求存在一个实际函数，其行为不能被功能测试宏改变。此外，同时提供 `xxx_locked()` 和 `xxx_unlocked()` 形式会导致混淆，即后缀是描述函数的行为还是应该使用它的情况。

提供了三个额外的例程 `flockfile()`、`ftrylockfile()` 和 `funlockfile()`（它们可能是宏），允许用户划定一系列同步执行的 I/O 语句。

`ungetc()` 函数相对于其他函数/宏的调用频率较低，因此不需要解锁变体。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- [2.5 标准 I/O 流 \(Standard I/O Streams\)](#)
- [flockfile\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [putc\(\)](#)
- [putchar\(\)](#)
-

变更历史 (CHANGE HISTORY)

首次发布于 Issue 5

包含在内以与 POSIX 线程扩展对齐。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

应用了 The Open Group 更正 U030/2，添加了应用程序使用部分，描述了应用程序应如何编写以避免函数作为宏实现时的情况。

Issue 7

`getc_unlocked()` 、 `getchar_unlocked()` 、 `putc_unlocked()` 和 `putchar_unlocked()` 函数从线程安全函数选项移至基础规范。

应用了 POSIX.1-2008，技术更正 1，XSH/TC1-2008/0232 [395]、XSH/TC1-2008/0233 [395]、XSH/TC1-2008/0234 [395] 和 XSH/TC1-2008/0235 [14]。

应用了 POSIX.1-2008，技术更正 2，XSH/TC2-2008/0151 [826]。

1.59. getchar

SYNOPSIS

```
#include <stdio.h>

int getchar(void);
```

DESCRIPTION

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 这卷文档遵循 ISO C 标准。

`getchar()` 函数应等价于 `getc(stdin)`。

RETURN VALUE

参考 `fgetc()`。

ERRORS

参考 `fgetc()`。

以下为提供信息的章节。

EXAMPLES

无。

APPLICATION USAGE

如果将 `getchar()` 返回的整数值存储到 `char` 类型的变量中，然后与整数常量 EOF 进行比较，这个比较可能永远不会成功，因为 `char` 类型变量在扩展为整数时的符号扩展是由实现定义的。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准I/O流
- `getc()`
- XBD `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008 Technical Corrigendum 1, XSH/TC1-2008/0236 [14]。

1.60. `getc_unlocked`, `getchar_unlocked`, `putc_unlocked`, `putchar_unlocked` — 带有显式客户 端锁定的 stdio 函数

概要

```
[CX] #include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

描述

应提供分别名为 `getc_unlocked()`、`getchar_unlocked()`、
`putc_unlocked()` 和 `putchar_unlocked()` 的 `getc()`、`getchar()`、
`putc()` 和 `putchar()` 函数版本，这些函数在功能上与原始版本等价，但它们不需要以完全线程安全的方式实现。当在 `flockfile()`（或 `ftrylockfile()`）和 `funlockfile()` 保护的范围内使用时，它们应该是线程安全的。这些函数可以安全地用于多线程程序中，当且仅当调用线程拥有 `(FILE *)` 对象时，就像在成功调用 `flockfile()` 或 `ftrylockfile()` 函数之后的情况。

如果 `getc_unlocked()` 或 `putc_unlocked()` 作为宏实现，它们可能多次计算 `stream`，因此 `stream` 参数永远不应该是有副作用的表达式。

返回值

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

错误

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

以下部分是提供信息的。

示例

无。

应用程序用法

由于它们可能作为宏实现，`getc_unlocked()` 和 `putc_unlocked()` 可能会错误处理有副作用的 *stream* 参数。特别是，`getc_unlocked(*f++)` 和 `putc_unlocked(c,*f++)` 不一定按预期工作。因此，在这种情况下使用这些函数之前应适当地执行以下语句：

```
#undef getc_unlocked
#undef putc_unlocked
```

原理

出于性能原因，一些 I/O 函数通常作为宏实现（例如 `putc()` 和 `getc()`）。为了安全，它们需要同步，但在每个字符上进行同步通常太昂贵了。尽管如此，人们认为安全问题更为重要；因此，要求 `getc()`、`getchar()`、`putc()` 和 `putchar()` 函数是线程安全的。然而，也提供了无锁版本，其名称清楚地表明其操作的不安全性，但可以利用它们的高性能。这些无锁版本只能在显式锁定程序区域内安全使用，使用导出的锁定原语。特别是，像这样的序列：

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

是允许的，并且会输出文本序列：

```
1
Line 2
```

而不会被其他线程的输出穿插。

让像 `getc()`、`putc()` 等标准名称映射到"更快但不安全"而不是"更慢但安全"的版本是错误的。无论哪种情况，在转换现有代码时，您仍然需要手动检查 `getc()`、`putc()` 等的所有使用。至少选择安全绑定作为默认会导致正确的代码，并保持"函数级别的原子性"不变量。否则会在转换的代码中引入不必要的同步错误。其他修改 `stdio` (`FILE *`) 结构或缓冲区的例程也被安全地同步。

注意，不需要 `getc_locked()`、`putc_locked()` 等形式的函数，因为这就是 `getc()`、`putc()` 等的功能。使用功能测试宏在 `getc_locked()` 和 `getc_unlocked()` 之间切换 `getc()` 的宏定义是不合适的，因为 ISO C 标准要求存在一个实际函数，其行为不能被功能测试宏改变。此外，同时提供 `xxx_locked()` 和 `xxx_unlocked()` 形式会导致关于后缀是描述函数行为还是应使用情况的混淆。

提供了三个额外的例程 `flockfile()`、`ftrylockfile()` 和 `funlockfile()`（它们可能是宏），以允许用户划定同步执行的 I/O 语句序列。

`ungetc()` 函数相对于其他函数/宏的调用频率较低，因此不需要无锁变体。

未来方向

无。

参见

- 2.5 标准 I/O 流
- `flockfile()`
- `getc()`
- `getchar()`
- `putc()`
- `putchar()`

XBD

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

应用了 The Open Group Corrigendum U030/2, 添加了描述应用程序应如何编写以避免函数作为宏实现的情况的应用程序用法。

Issue 7

`getc_unlocked()` 、 `getchar_unlocked()` 、 `putc_unlocked()` 和 `putchar_unlocked()` 函数从线程安全函数选项移动到基础部分。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], 和 XSH/TC1-2008/0235 [14]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826]。

信息性文本结束。

1.61. getenv, secure_getenv — 获取环境变量的值

概要

```
#include <stdlib.h>

char *getenv(const char *name);

[CX] char *secure_getenv(const char *name);
```

描述

[CX] 本参考页描述的功能与ISO C标准保持一致。此处描述的要求与ISO C标准之间的任何冲突都是无意的。本POSIX.1-2024版本遵循ISO C标准。

`getenv()` 函数应在调用进程的环境中搜索环境变量 *name* (见XBD 8. 环境变量)，如果存在则返回指向该环境变量值的指针。如果找不到指定的环境变量，则应返回空指针。应用程序应确保不修改 `getenv()` 函数指向的字符串，[CX] 除非该字符串是先前通过给 `environ` [XSI] 赋值或使用 `putenv()` 而放入环境中的可修改对象的一部分。

[CX] `getenv()` 返回的指针应指向 `environ` 指向的环境数据中的字符串。

注意：

这个要求是对ISO C标准的扩展，ISO C标准允许 `getenv()` 将数据复制到内部缓冲区。

`secure_getenv()` 函数应等同于 `getenv()`，但如果调用进程不满足以下所有安全条件，则应返回空指针：

1. 调用进程的有效用户ID和实际用户ID在程序启动时相等。
2. 调用进程的有效组ID和实际组ID在程序启动时相等。
3. 实现定义的额外安全条件。

返回值

成功完成后, `getenv()` 应返回指向包含指定 *name* 值的字符串的指针。如果在调用进程的环境中找不到指定的 *name*, 则应返回空指针。

[CX] 成功完成后, `secure_getenv()` 应返回指向包含指定 *name* 值的字符串的指针。如果在调用进程的环境中找不到指定的 *name*, 或者调用进程不满足描述中列出的安全条件, 则应返回空指针。

错误

未定义任何错误。

示例

获取环境变量的值

以下示例获取HOME环境变量的值。

```
#include <stdlib.h>
...
const char *name = "HOME";
char *value;

value = getenv(name);
```

应用程序使用

无。

原理说明

`clearenv()` 函数曾被考虑但被拒绝。`putenv()` 函数现已包含在内, 以与 Single UNIX Specification 保持一致。

本标准的某些早期版本不要求 `getenv()` 是线程安全的，因为它被允许返回指向内部缓冲区的值。然而，ISO C标准允许的这种行为在POSIX.1中不再被允许。POSIX.1 要求数据环境通过 `environ[]` 可用，因此没有理由不让 `getenv()` 返回指向实际数据的指针而不是副本。因此，现在要求 `getenv()` 是线程安全的（除非另一个线程修改了环境）。

符合要求的应用程序被要求不直接修改 `environ` 指向的指针，而是只使用 `setenv()`、`unsetenv()` 和 `putenv()` 函数，或对 `environ` 本身赋值来操作进程环境。这个约束允许实现正确管理它分配的内存。这使得实现在调用 `unsetenv()` 时可以释放已分配给存储在 `environ` 中的字符串（以及可能指向它们的指针）的任何空间。C运行时启动过程（调用 `main()` 并可能初始化 `environ` 的过程）也可以初始化一个标志，表示环境尚未复制到分配的存储空间，或者单独的表尚未初始化。如果应用程序通过给 `environ` 赋新值切换到完整的新环境，这可以被 `getenv()`、`setenv()`、`unsetenv()` 或 `putenv()` 检测到，实现此时可以基于新环境重新初始化。（这可能包括将环境字符串复制到新数组并让 `environ` 指向它。）

事实上，为了获得更高的 `getenv()` 性能，不提供 `putenv()` 的实现也可以在单独的数据结构中维护环境的单独副本，该结构可以更快地搜索（如索引哈希表或二叉树），并在调用 `setenv()` 或 `unsetenv()` 时同时更新它和 `environ` 上的线性列表。在提供 `putenv()` 的实现上，这样的副本可能仍然有价值，但需要考虑应用程序可以直接修改使用 `putenv()` 添加的环境字符串内容的事实。例如，如果通过搜索副本找到的环境字符串是使用 `putenv()` 添加的，实现需要检查 `environ` 中的字符串是否仍然具有相同的名称（和值，如果副本包含值），并且每当搜索副本没有匹配项时，实现需要搜索 `environ` 中每个使用 `putenv()` 添加的环境字符串，以防它们中有任何更改了名称现在匹配。因此，每次使用 `putenv()` 添加到环境都会降低拥有副本的速度优势。

对于拥有大量环境变量的应用程序，`getenv()` 的性能可能很重要。通常，这样的应用程序将环境用作用户可配置参数的资源数据库。这些变量位于用户 shell 环境中的事实通常意味着任何其他使用环境变量的程序（如尝试使用 `COLUMNS` 的 `ls`，或者几乎所有实用程序（`LANG`、`LC_ALL` 等））都会因为线性搜索变量而同样变慢。

维护单独的数据结构，甚至管理其消耗内存的实现目前不是必需的，因为被认为会减少不想改变其历史实现的实现者之间的共识。

未来方向

无。

参见

- `exec()`
- `putenv()`
- `setenv()`
- `unsetenv()`

XBD 8. 环境变量, `<stdlib.h>`

变更历史

首次在Issue 1中发布。源自SVID的Issue 1。

Issue 5

先前在应用程序使用部分的规范性文本被移至返回值部分。

在描述中添加了一个注释，指出此函数不需要是可重入的。

Issue 6

进行了以下更改以与IEEE P1003.1a草案标准保持一致：

- 添加了对新的 `setenv()` 和 `unsetenv()` 函数的引用。

规范性文本被更新以避免对应用程序要求使用"must"一词。

Issue 7

应用了Austin Group解释1003.1-2001 #062，澄清了调用 `putenv()` 也可能导致字符串被覆盖。

应用了Austin Group解释1003.1-2001 #148，添加了未来方向。

应用了Austin Group解释1003.1-2001 #156。

应用了POSIX.1-2008技术勘误1，XSH/TC1-2008/0238 [75,428]、XSH/TC1-2008/0239 [167]和XSH/TC1-2008/0240 [167]。

应用了POSIX.1-2008技术勘误2，XSH/TC2-2008/0157 [656]。

Issue 8

应用了Austin Group缺陷188和1394，将 `getenv()` 更改为线程安全的。

应用了Austin Group缺陷922，添加了 `secure_getenv()` 函数。

1.62. gets

概要

```
#include <stdio.h>

char *gets(char *s);
```

描述

本参考页描述的功能与 ISO C 标准保持一致。如果此处描述的要求与 ISO C 标准之间存在冲突，则非有意为之。POSIX.1-2017 卷遵循 ISO C 标准。

`gets()` 函数应从标准输入流 `stdin` 读取字节到 `s` 指向的数组中，直到读取到 `<newline>`（换行符）或遇到文件结束条件。任何 `<newline>` 都将被丢弃，并在读取到数组中的最后一个字节之后立即放置一个空字节。

`gets()` 函数可以标记与流关联的文件的最后数据访问时间戳以进行更新。第一次成功执行使用 `stream` 的 `fgetc()`、`fgets()`、`fread()`、`fscanf()`、`getc()`、`getchar()`、`getdelim()`、`getline()`、`gets()` 或 `scanf()` 函数时，应标记最后数据访问时间戳以进行更新，这些函数返回的数据不是由先前调用 `ungetc()` 提供的。

返回值

成功完成时，`gets()` 应返回 `s`。如果流的文件结束指示符已设置，或者流处于文件结束位置，则应设置流的文件结束指示符并且 `gets()` 应返回空指针。如果发生读取错误，则应设置流的错误指示符，`gets()` 应返回空指针，并设置 `errno` 以指示错误。

错误

参考 `fgetc()`。

以下部分为补充信息。

示例

无。

应用程序使用

读取溢出 `s` 指向的数组的行会导致未定义行为。建议使用 `fgets()`。

由于用户无法指定传递给 `gets()` 的缓冲区长度，不鼓励使用此函数。读取的字符串长度是无限的。可能以导致应用程序失败或可能的系统安全违规的方式溢出此缓冲区。

应用程序应使用 `fgets()` 函数而不是已过时的 `gets()` 函数。

原理

标准开发者决定将 `gets()` 函数标记为过时，即使它在 ISO C 标准中，这是由于缓冲区溢出的可能性。

未来方向

`gets()` 函数可能在未来的版本中被移除。

参见

- [标准 I/O 流](#)
- `feof()`
- `ferror()`
- `fgets()`
-

更改历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #051，阐明了返回值部分。

`gets()` 函数被标记为过时。

进行了与细粒度时间戳支持相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0257 [14]。

补充信息结束。

1.63. gmtime, gmtime_r — 将时间值转换为分解的 UTC时间

SYNOPSIS

```
#include <time.h>

struct tm *gmtime(const time_t *timer);

[CX] struct tm *gmtime_r(const time_t *restrict timer,
                         struct tm *restrict result);
```

DESCRIPTION

对于 `gmtime()` :

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

`gmtime()` 函数应将由 `timer` 指向的自纪元以来以秒为单位的时间转换为分解时间，以协调世界时 (UTC) 表示。

[CX] 用作 `gmtime()` 参数的自纪元以来以秒为单位的时间与 `tm` 结构 (在 `<time.h>` 头文件中定义) 之间的关系是：结果应按照自纪元以来秒数定义中给出的表达式指定 (参见 XBD 4.19 自纪元以来的秒数)，其中结构中的名称与表达式中的名称相对应。

同样的关系适用于 `gmtime_r()` 。

`gmtime()` 函数不必是线程安全的；但是，`gmtime()` 应避免与其自身、`asctime()`、`ctime()` 和 `localtime()` 之外的所有函数发生数据竞争。

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应返回两个静态对象之一的值：一个分解时间结构和一个 `char` 类型的数组。执行任何返回指向这些对象类型之一指针的函数都可能覆盖通过先前对其中任何函数的调用返回的值所指向的任何相同类型对象中的信息。

[CX] `gmtime_r()` 函数应将由 `timer` 指向的自纪元以来以秒为单位的时间转换为以协调世界时 (UTC) 表示的分解时间。分解时间存储在由 `result` 引用的结构中。`gmtime_r()` 函数还应返回同一结构的地址。

RETURN VALUE

成功完成后, `gmtime()` 函数应返回指向 `struct tm` 的指针。如果检测到错误, `gmtime()` 应返回空指针 [CX] 并设置 `errno` 以指示错误。

成功完成后, `gmtime_r()` 应返回由参数 `result` 指向的结构地址。结构的 `tm_zone` 成员应设置为指向字符串 "UTC" 的指针, 该字符串应具有静态存储持续时间。如果检测到错误, `gmtime_r()` 应返回空指针并设置 `errno` 以指示错误。

ERRORS

`gmtime()` [CX] 和 `gmtime_r()` 函数在以下情况下应失败:

- [EOVERFLOW] [CX] 结果无法表示。

EXAMPLES

无。

APPLICATION USAGE

`gmtime_r()` 函数是线程安全的, 并在用户提供的缓冲区中返回值, 而不是可能使用每次调用都可能被覆盖的静态数据区域。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `asctime()`

- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`

XBD 4.19 自纪元以来的秒数, `<time.h>`

CHANGE HISTORY

首次发布于 Issue 1

源自 SVID 的 Issue 1。

Issue 5

- 在 DESCRIPTION 中添加了一个注释, 指出 `gmtime()` 函数不必是可重入的。
- 为了与 POSIX 线程扩展保持一致, 包含了 `gmtime_r()` 函数。

Issue 6

- `gmtime_r()` 函数被标记为线程安全函数选项的一部分。
- 标记了超出 ISO C 标准的扩展。
- 更新了 APPLICATION USAGE 部分, 添加了关于线程安全函数及其避免可能使用静态数据区域的注释。
- 为了与 ISO/IEC 9899:1999 标准保持一致, 向 `gmtime_r()` 原型添加了 `restrict` 关键字。
- 应用了 IEEE Std 1003.1-2001/Cor 1-2002, 项目 XSH/TC1/D6/27, 添加了 [OVERFLOW] 错误。

- 应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/48, 更新了 `gmtime_r()` 的错误处理。

Issue 7

- 应用了 Austin Group Interpretation 1003.1-2001 #156。
- `gmtime_r()` 函数从线程安全函数选项移至 Base。

Issue 8

- 应用了 Austin Group Defect 1302, 使 `gmtime()` 函数与 ISO/IEC 9899:2018 标准保持一致。
 - 应用了 Austin Group Defect 1376, 从某些源自 ISO C 标准的文本中移除了 CX 着色, 并更新以匹配 ISO C 标准。
 - 应用了 Austin Group Defect 1533, 向 `tm` 结构添加了 `tm_gmtoff` 和 `tm_zone`。
-

1.64. gmtime, gmtime_r — 将时间值转换为分解的 UTC时间

概要

```
#include <time.h>

struct tm *gmtime(const time_t *timer);

[CX] struct tm *gmtime_r(const time_t *restrict timer,
                         struct tm *restrict result);
```

描述

对于 `gmtime()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

`gmtime()` 函数应将 `timer` 指向的自纪元 (Epoch) 以来以秒为单位的时间转换为分解时间，表示为协调世界时 (UTC)。

[CX] 作为 `gmtime()` 参数使用的自纪元以来以秒为单位的时间与 `tm` 结构 (在 `<time.h>` 头文件中定义) 之间的关系是，结果应符合自纪元以来秒数定义中给出的表达式 (见 XBD 4.19 自纪元以来的秒数)，其中结构中的名称与表达式中的名称相对应。

同样的关系也适用于 `gmtime_r()`。

`gmtime()` 函数不必是线程安全的；但是，`gmtime()` 应避免与除自身、`asctime()`、`ctime()` 和 `localtime()` 之外的所有函数发生数据竞争。

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解时间结构和一个 `char` 类型的数组。执行返回指向这些对象类型之一指针的任何函数都可能覆盖任何先前调用其中任何函数返回的值所指向的相同类型对象中的信息。

[CX] `gmtime_r()` 函数应将 `timer` 指向的自纪元以来以秒为单位的时间转换为表示为协调世界时 (UTC) 的分解时间。分解时间存储在 `result` 引用的结构中。`gmtime_r()` 函数还应返回相同结构的地址。

返回值

成功完成后，`gmtime()` 函数应返回指向 `struct tm` 的指针。如果检测到错误，`gmtime()` 应返回空指针 [CX] 并设置 `errno` 以指示错误。

成功完成后，`gmtime_r()` 应返回参数 `result` 指向的结构地址。结构的 `tm_zone` 成员应设置为指向字符串 "UTC" 的指针，该字符串应具有静态存储持续时间。如果检测到错误，`gmtime_r()` 应返回空指针并设置 `errno` 以指示错误。

错误

`gmtime()` [CX] 和 `gmtime_r()` 函数在以下情况下应失败：

- [EOVERFLOW] [CX] 结果无法表示。

示例

无。

应用程序用法

`gmtime_r()` 函数是线程安全的，在用户提供的缓冲区中返回值，而不是可能使用每次调用可能被覆盖的静态数据区域。

原理

无。

未来方向

无。

参见

`asctime()` , `clock()` , `ctime()` , `difftime()` , `futimens()` ,
`localtime()` , `mktime()` , `strftime()` , `strptime()` , `time()`

XBD 4.19 自纪元以来的秒数, `<time.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

在描述中添加了指出 `gmtime()` 函数不必是可重入的注释。

包含 `gmtime_r()` 函数以与 POSIX 线程扩展对齐。

Issue 6

`gmtime_r()` 函数被标记为线程安全函数选项的一部分。

超出 ISO C 标准的扩展被标记。

应用程序用法部分更新，增加了关于线程安全函数及其避免可能使用静态数据区域的注释。

为与 ISO/IEC 9899:1999 标准对齐，向 `gmtime_r()` 原型添加了 `restrict` 关键字。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/27，添加了 [EOVERFLOW] 错误。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/48，更新了 `gmtime_r()` 的错误处理。

Issue 7

应用 Austin Group 解释 1003.1-2001 #156。

`gmtime_r()` 函数从线程安全函数选项移动到基础部分。

Issue 8

应用 Austin Group 缺陷 1302，使 `gmtime()` 函数与 ISO/IEC 9899:2018 标准对齐。

应用 Austin Group 缺陷 1376，从源自 ISO C 标准的某些文本中移除 CX 着色并更新以匹配 ISO C 标准。

应用 Austin Group 缺陷 1533，向 `tm` 结构添加了 `tm_gmtoff` 和 `tm_zone`。

1.65. `imaxabs` — 返回绝对值

概要

```
#include <inttypes.h>

intmax_t imaxabs(intmax_t j);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`imaxabs()` 函数应计算整数 `j` 的绝对值。如果结果无法表示，则行为未定义。

返回值

`imaxabs()` 函数应返回绝对值。

错误

未定义错误。

示例

无。

应用用法

由于 POSIX.1 要求 `intmax_t` 使用二进制补码表示，具有最大量级的负整数 `{INTMAX_MIN}` 的绝对值无法表示，因此 `imaxabs(INTMAX_MIN)` 是未定义的。

原理

无。

未来方向

无。

参见

- `imaxdiv()`
- XBD `<inttypes.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 8

应用了 Austin Group Defect 1108，更改了 APPLICATION USAGE 部分。

1.66. imaxdiv

SYNOPSIS

```
#include <inttypes.h>

imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`imaxdiv()` 函数应在单个操作中计算 `numer / denom` 和 `numer % denom`。

RETURN VALUE

`imaxdiv()` 函数应返回一个 `imaxdiv_t` 类型的结构体，包含商和余数两部分。该结构体应包含（顺序不限）成员 `quot`（商）和 `rem`（余数），每个成员的类型均为 `intmax_t`。

如果结果的任何部分无法表示，则行为是未定义的。

ERRORS

未定义错误。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [imaxabs\(\)](#)
- XBD [<inttypes.h>](#)

CHANGE HISTORY

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

1.67. isalnum, isalnum_l — 测试字母数字字符

概要

```
#include <ctype.h>

int isalnum(int c);

[CX] int isalnum_l(int c, locale_t locale);
```

描述

对于 `isalnum()`：[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`isalnum()` [CX] 和 `isalnum_l()` 函数应测试 `c` 是否在当前区域设置中属于 **alpha** 或 **digit** 类的字符，[CX] 或分别在由 `locale` 表示的区域设置中；参见 XBD 7. 区域设置。

`c` 参数是一个 `int`，其值应确保应用程序可以表示为 `unsigned char` 或等于宏 EOF 的值。如果参数具有任何其他值，则行为是未定义的。

[CX] 如果 `isalnum_l()` 的 `locale` 参数是特殊区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为是未定义的。

返回值

如果 `c` 是字母数字字符，`isalnum()` [CX] 和 `isalnum_l()` 函数应返回非零值；否则，它们应返回 0。

错误

未定义任何错误。

以下部分是提供信息的。

示例

无。

应用程序用法

为确保应用程序的可移植性，特别是在不同自然语言之间，应仅使用这些函数和参见 also 部分列出的参考页面中的函数进行字符分类。

原理

无。

未来方向

无。

参见

- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，以避免使用术语"must"来表示应用程序要求。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `isalnum_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0274 [302], XSH/TC1-2008/0275 [283], 和 XSH/TC1-2008/0276 [283]。

1.68. isalpha, isalpha_l — 测试字母字符

概要

```
#include <ctype.h>

int isalpha(int c);

[CX] int isalpha_l(int c, locale_t locale);
```

描述

对于 `isalpha()`：[CX] 本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 版本遵循 ISO C 标准。

`isalpha()` [CX] 和 `isalpha_l()` 函数应测试 `c` 是否为当前区域设置中 `alpha` 字符类的字符，[CX] 或分别为 `locale` 表示的区域设置中的字符；参见 XBD 7. 区域设置。

`c` 参数是一个 `int` 类型，应用程序应确保其值可以表示为 `unsigned char` 或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `isalpha_l()` 的 `locale` 参数是特殊区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

返回值

如果 `c` 是字母字符，`isalpha()` [CX] 和 `isalpha_l()` 函数应返回非零值；否则应返回 0。

错误

未定义任何错误。

以下部分为提供信息的内容。

示例

无。

应用程序用法

为确保应用程序的可移植性，尤其是在自然语言之间，仅应使用这些函数以及参考部分中列出的其他函数进行字符分类。

基本原理

无。

未来方向

无。

另请参阅

- [isalnum\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

变更历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新, 以避免对应用程序要求使用"必须"一词。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `isalpha_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0277 [302], XSH/TC1-2008/0278 [283], 和 XSH/TC1-2008/0279 [283]。

1.69. isblank, isblank_l — 测试空白字符

概要

```
#include <ctype.h>

int isblank(int c);

[CX] int isblank_l(int c, locale_t locale);
```

描述

对于 `isblank()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 卷遵循 ISO C 标准。

`isblank()` [CX] 和 `isblank_l()` 函数应测试 `c` 是否为当前语言环境中 **blank** (空白) 类别的字符，[CX] 或分别测试 `locale` 表示的语言环境中的字符；参见 XBD [7. Locale]。

`c` 参数是 `int` 类型，应用程序应确保其值是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为是未定义的。

[CX] 如果传递给 `isblank_l()` 的 `locale` 参数是特殊语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为是未定义的。

返回值

如果 `c` 是 (空白字符)，`isblank()` [CX] 和 `isblank_l()` 函数应返回非零值；否则应返回 0。

错误

未定义错误。

示例

无。

应用程序使用

为确保应用程序的可移植性，特别是在不同自然语言之间，应仅使用这些函数和"参见"部分列出的参考页面中的函数进行字符分类。

原理

无。

未来方向

无。

参见

`isalnum()`、
`isalpha()`、
`iscntrl()`、
`isdigit()`、
`isgraph()`、
`islower()`、
`isprint()`、
`ispunct()`、
`isspace()`、
`isupper()`、
`isxdigit()`、
`setlocale()`、
`uselocale()`

XBD [7. Locale]、`<ctype.h>`、`<locale.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `isblank_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0280 [302], XSH/TC1-2008/0281 [283] 和 XSH/TC1-2008/0282 [283]。

1.70. iscntrl, iscntrl_l - 测试控制字符

SYNOPSIS

```
#include <ctype.h>

int iscntrl(int c);

[CX] int iscntrl_l(int c, locale_t locale);
```

DESCRIPTION

对于 `iscntrl()`：[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`iscntrl()` [CX] 和 `iscntrl_l()` 函数应测试 `c` 是否在当前语言环境中 [CX] 或分别由 `locale` 表示的语言环境中属于 `cntrl` 类字符；参见 XBD [7. Locale]。

`c` 参数的类型为 `int`，应用程序应确保其值是一个可表示为 `unsigned char` 的字符，或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `iscntrl_l()` 的 `locale` 参数是特殊的语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为未定义。

RETURN VALUE

如果 `c` 是控制字符，`iscntrl()` [CX] 和 `iscntrl_l()` 函数应返回非零值；否则，它们应返回 0。

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

为确保应用程序的可移植性，特别是在不同自然语言环境之间，字符分类只能使用这些函数以及 SEE ALSO 部分列出的参考页面中的函数。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

更新规范性文本，避免对应用程序要求使用"必须"一词。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `iscntrl_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0283 [302], XSH/TC1-2008/0284 [283] 和 XSH/TC1-2008/0285 [283]。

1.71. isdigit, isdigit_l — 测试十进制数字字符

概要

```
#include <ctype.h>

int isdigit(int c);

[CX] int isdigit_l(int c, locale_t locale);
```

描述

对于 `isdigit()` :

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 的本卷遵循 ISO C 标准。

`isdigit()` [CX] 和 `isdigit_l()` 函数应测试 `c` 是否为当前语言环境 [CX] 或由 `locale` 表示的语言环境中的 `digit` 类字符；参见 XBD 7. Locale。

`c` 参数是一个 `int`，应用程序应确保其值是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `isdigit_l()` 的 `locale` 参数是特殊语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为未定义。

返回值

如果 `c` 是十进制数字字符，`isdigit()` [CX] 和 `isdigit_l()` 函数应返回非零值；否则应返回 0。

错误

未定义错误。

以下章节为补充信息。

示例

无。

应用用法

为确保应用程序的可移植性，特别是在自然语言之间，应仅使用这些函数和 SEE ALSO 部分列出的参考页面中的函数进行字符分类。

原理

无。

未来方向

无。

参见

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)

XBD [7. Locale](#), ,

更改历史

首次发布于 Issue 1。

源自 SVID 的 Issue 1。

Issue 6

更新规范性文本，避免对应用程序要求使用术语"必须"。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `isdigit_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0286 [302], XSH/TC1-2008/0287 [283] 和 XSH/TC1-2008/0288 [283]。

补充信息结束。

1.72. `isgraph`, `isgraph_l` — 测试可见字符

概要

```
#include <ctype.h>

int isgraph(int c);

[CX] int isgraph_l(int c, locale_t locale);
```

描述

对于 `isgraph()`：[CX] 本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`isgraph()` [CX] 和 `isgraph_l()` 函数应测试 `c` 在当前语言环境中，[CX] 或在由 `locale` 表示的语言环境中是否为 **graph** 字符类；参见 XBD 7. 语言环境。

`c` 参数是一个 **int** 类型，应用程序应确保其值是可以表示为 **unsigned char** 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `isgraph_l()` 的 `locale` 参数是特殊语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为未定义。

返回值

`isgraph()` [CX] 和 `isgraph_l()` 函数应在 `c` 是具有可见表示形式的字符时返回非零值；否则，它们应返回 0。

错误

未定义任何错误。

以下各节为信息性内容。

示例

无。

应用程序用法

为确保应用程序的可移植性，特别是在不同自然语言之间，只能使用这些函数和参见部分中列出的参考页中的函数来进行字符分类。

原理

无。

未来方向

无。

参见

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，避免对应用程序要求使用术语"必须" (must)。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `isgraph_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0289 [302], XSH/TC1-2008/0290 [283], 和 XSH/TC1-2008/0291 [283]。

信息性文本结束。

1.73. islower, islower_l - 测试小写字母

SYNOPSIS

```
#include <ctype.h>

int islower(int c);

/* XSI 扩展 */
int islower_l(int c, locale_t locale);
```

DESCRIPTION

对于 `islower()`：本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 的这一卷遵循 ISO C 标准。

`islower()` 和 `islower_l()` 函数应测试 `c` 是否分别为当前区域设置或由 `locale` 表示的区域设置中的 **lower** 类字符；参见 XBD 7. Locale。

`c` 参数是一个 `int`，其值应为应用程序可确保表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

如果 `islower_l()` 的 `locale` 参数是特殊区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

RETURN VALUE

如果 `c` 是小写字母，`islower()` 和 `islower_l()` 函数应返回非零值；否则，它们应返回 0。

ERRORS

未定义错误。

EXAMPLES

测试小写字母

下面是两个示例，第一个使用 `islower()`，第二个使用多个并发区域设置和 `islower_l()`。

这些示例基于当前区域设置测试值是否为小写字母，然后将其用作键值的一部分。

```
/* 示例 1 -- 使用 islower() */
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>

...
char *keystr;
int elementlen, len;
unsigned char c;

...
setlocale(LC_ALL, "");
...

len = 0;
while (len < elementlen) {
    c = (unsigned char) (rand() % 256);
    ...
    if (islower(c))
        keystr[len++] = c;
}
...
...
```

```
/* 示例 2 -- 使用 islower_l() */
#include <ctype.h>
#include <stdlib.h>
#include <locale.h>

...
char *keystr;
int elementlen, len;
unsigned char c;

...
locale_t loc = newlocale (LC_ALL_MASK, "", (locale_t) 0);
...
len = 0;
while (len < elementlen) {
    c = (unsigned char) (rand() % 256);
    ...
    if (islower_l(c, loc))
        keystr[len++] = c;
}
```

```
    }  
...
```

APPLICATION USAGE

为确保应用程序的可移植性，特别是在不同语言之间，只能使用这些函数和 SEE ALSO 部分列出的参考页面中的函数进行字符分类。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [isprint\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [uselocale\(\)](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，避免在应用程序要求中使用术语"must"，并添加了示例。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `islower_l()` 函数。

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0292 [302], XSH/TC1-2008/0293 [283], XSH/TC1-2008/0294 [283], XSH/TC1-2008/0295 [302], and XSH/TC1-2008/0296 [304] 已应用。

1.74. isprint, isprint_l — 测试可打印字符

SYNOPSIS

```
#include <ctype.h>

int isprint(int c);

/* [CX] 扩展 */
int isprint_l(int c, locale_t locale);
```

DESCRIPTION

对于 `isprint()` :

[本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非预期的。本卷 POSIX.1-2024 遵从 ISO C 标准。]

`isprint()` 和 `isprint_l()` 函数应测试 `c` 是否分别为当前语言环境或由 `locale` 表示的语言环境中的 **print** 类字符；参见 XBD [7. Locale]。

`c` 参数是一个 `int`，其值应由应用程序确保是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[如果 `isprint_l()` 的 `locale` 参数是特殊语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为未定义。]

RETURN VALUE

如果 `c` 是可打印字符，`isprint()` 和 `isprint_l()` 函数应返回非零值；否则，它们应返回 0。

ERRORS

未定义任何错误。

APPLICATION USAGE

为确保应用程序的可移植性，特别是在不同自然语言之间，应仅使用这些函数和 SEE ALSO 部分所列参考页面中的函数进行字符分类。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [ispunct\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

XBD [7. Locale], [`<ctype.h>`](#), [`<locale.h>`](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范文本已更新，以避免在应用程序要求中使用"必须"一词。

Issue 7

`isprint_l()` 函数从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0297 [302], XSH/TC1-2008/0298 [283], 和 XSH/TC1-2008/0299 [283]。

1.75. `ispunct`, `ispunct_l` — 测试标点符号字符

SYNOPSIS

```
#include <ctype.h>

int ispunct(int c);

[CX] int ispunct_l(int c, locale_t locale);
```

DESCRIPTION

对于 `ispunct()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`ispunct()` [CX] 和 `ispunct_l()` 函数应测试 `c` 是否为当前区域设置中的 **punct** 字符类别，[CX] 或分别为 `locale` 所代表的区域设置中的 **punct** 字符类别；参见 XBD 7. 区域设置。

`c` 参数是一个 `int`，应用程序应确保其值是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `ispunct_l()` 的 `locale` 参数是特殊区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

RETURN VALUE

如果 `c` 是标点符号，`ispunct()` [CX] 和 `ispunct_l()` 函数应返回非零值；否则，它们应返回 0。

ERRORS

未定义任何错误。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

为确保应用程序的可移植性，特别是在不同自然语言之间，应仅使用这些函数和 SEE ALSO 章节中列出的参考页中的函数进行字符分类。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [isalnum\(\)](#)
- [isalpha\(\)](#)
- [isblank\(\)](#)
- [iscntrl\(\)](#)
- [isdigit\(\)](#)
- [isgraph\(\)](#)
- [islower\(\)](#)
- [isprint\(\)](#)
- [isspace\(\)](#)
- [isupper\(\)](#)
- [isxdigit\(\)](#)
- [setlocale\(\)](#)
- [use locale\(\)](#)

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，以避免在应用程序要求中使用"必须"一词。

Issue 7

`ispunct_l()` 函数从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0300 [302], XSH/TC1-2008/0301 [283], 和 XSH/TC1-2008/0302 [283]。

1.76. isspace, isspace_l — 测试空白字符

SYNOPSIS

```
#include <ctype.h>

int isspace(int c);

[CX] int isspace_l(int c, locale_t locale);
```

DESCRIPTION

对于 `isspace()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`isspace()` [CX] 和 `isspace_l()` 函数应测试 `c` 是否在当前区域环境中属于 **space** 字符类别，[CX] 或分别在由 `locale` 表示的区域环境中属于该字符类别；参见 XBD 7. Locale。

`c` 参数是一个 `int`，应用程序应确保其值是一个可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `isspace_l()` 的 `locale` 参数是特殊的区域对象 `LC_GLOBAL_LOCALE` 或不是有效的区域对象句柄，则行为未定义。

RETURN VALUE

如果 `c` 是空白字符，`isspace()` [CX] 和 `isspace_l()` 函数应返回非零值；否则应返回 0。

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

为确保应用程序的可移植性，特别是在不同自然语言之间，应仅使用这些函数和 SEE ALSO 部分列出的参考页中的函数进行字符分类。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

`isalnum()` , `isalpha()` , `iscntrl()` , `isdigit()` , `isgraph()` ,
`islower()` , `isprint()` , `ispunct()` , `isupper()` , `isxdigit()` ,
`setlocale()` , `uselocale()`

XBD 7. Locale, `<ctype.h>` , `<locale.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

更新规范性文本以避免对应用程序要求使用术语"must"。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `isspace_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0303 [302], XSH/TC1-2008/0304 [283], 和 XSH/TC1-2008/0305 [283]。

1.77. isupper, isupper_l — 测试大写字母

概要

```
#include <ctype.h>

int isupper(int c);

[CX] int isupper_l(int c, locale_t locale);
```

描述

对于 `isupper()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵从 ISO C 标准。

`isupper()` [CX] 和 `isupper_l()` 函数应测试 `c` 在当前区域设置中，[CX] 或在由 `locale` 表示的区域设置中是否为 `upper` 类字符；参见 XBD 7. 区域设置。

`c` 参数是一个 `int`，其值应由应用程序确保是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值，则行为未定义。

[CX] 如果 `isupper_l()` 的 `locale` 参数是特殊区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

返回值

如果 `c` 是大写字母，`isupper()` [CX] 和 `isupper_l()` 函数应返回非零值；否则，它们应返回 0。

错误

未定义错误。

示例

无。

应用程序用法

为确保应用程序的可移植性，尤其是在不同自然语言之间，只应使用这些函数和参见部分所列参考页中的函数进行字符分类。

原理

无。

未来方向

无。

参见

`isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`,
`isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`,
`isxdigit()`, `setlocale()`, `uselocale()`

XBD 7. 区域设置, `<ctype.h>`, `<locale.h>`

变更历史

首次发布于 Issue 1。派生自 SVID 的 Issue 1。

Issue 6

规范性文本已更新，以避免使用术语"must"来表示应用程序要求。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `isupper_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0306 [302], XSH/TC1-2008/0307 [283], 和 XSH/TC1-2008/0308 [283]。

1.78. isxdigit, isxdigit_l — 测试十六进制数字字符

概要

```
#include <ctype.h>

int isxdigit(int c);

[CX] int isxdigit_l(int c, locale_t locale);
```

描述

对于 `isxdigit()` :

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`isxdigit()` [CX] 和 `isxdigit_l()` 函数应测试 `c` 在当前语言环境中是否为 **xdigit** 类的字符, [CX] 或分别在由 `locale` 表示的语言环境中是否为 **xdigit** 类的字符; 参见 XBD 7. 语言环境。

`c` 参数是一个 `int`, 应用程序应确保其值是可表示为 `unsigned char` 的字符或等于宏 EOF 的值。如果参数具有任何其他值, 则行为是未定义的。

[CX] 如果 `isxdigit_l()` 的 `locale` 参数是特殊的语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄, 则行为是未定义的。

返回值

如果 `c` 是十六进制数字字符, `isxdigit()` [CX] 和 `isxdigit_l()` 函数应返回非零值; 否则, 它们应返回 0。

错误

未定义错误。

以下章节为信息性内容。

示例

无。

应用程序用法

为确保应用程序的可移植性，尤其是在不同自然语言之间，应仅使用这些函数和 SEE ALSO 部分列出的参考页中的函数来进行字符分类。

原理

无。

未来方向

无。

另请参阅

`isalnum()` , `isalpha()` , `isblank()` , `iscntrl()` , `isdigit()` ,
`isgraph()` , `islower()` , `isprint()` , `ispunct()` , `isspace()` ,
`isupper()`

XBD 7. 语言环境, `<ctype.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

更新规范性文本，避免对应用程序要求使用术语"必须" (must)。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `isxdigit_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0347 [302], XSH/TC1-2008/0348 [283], 和 XSH/TC1-2008/0349 [283]。

信息性文本结束。

1.79. kill

SYNOPSIS

```
#include <signal.h>

int kill(pid_t pid, int sig);
```

DESCRIPTION

`kill()` 函数应向由 `pid` 指定的进程或进程组发送信号。要发送的信号由 `sig` 指定，可以是 `<signal.h>` 中列出的信号之一，也可以是 0。如果 `sig` 为 0（空信号），则执行错误检查但不实际发送信号。空信号可用于检查 `pid` 的有效性。

要使进程有权向由 `pid` 指定的进程发送信号，除非发送进程具有适当的权限，否则发送进程的实际或有效用户 ID 必须与接收进程的实际或已保存的设置用户 ID 匹配。

如果 `pid` 大于 0，`sig` 应发送给进程 ID 等于 `pid` 的进程。

如果 `pid` 为 0，`sig` 应发送给所有进程组 ID 等于发送者进程组 ID 且进程有权限发送信号的进程（不包括一组未指定的系统进程）。

如果 `pid` 为 -1，`sig` 应发送给进程有权限发送该信号的所有进程（不包括一组未指定的系统进程）。

如果 `pid` 为负数但不是 -1，`sig` 应发送给所有进程组 ID 等于 `pid` 绝对值且进程有权限发送信号的进程（不包括一组未指定的系统进程）。

如果 `pid` 的值导致为发送进程生成 `sig`，并且如果 `sig` 对于调用线程未被阻塞，并且如果没有其他线程解除对 `sig` 的阻塞或在 `sigwait()` 函数中等待 `sig`，那么在 `kill()` 返回之前，`sig` 或至少一个待处理的未阻塞信号应被传递给发送线程。

当向与发送进程属于同一会话的进程发送 SIGCONT 时，不应应用上述用户 ID 检查。

提供扩展安全控制的实现可能会对信号发送（包括空信号）施加进一步的实现定义的限制。特别是，系统可能拒绝 `pid` 指定的部分或全部进程的存在。

如果进程有权限向 `pid` 指定的任何进程发送 `sig`，则 `kill()` 函数成功。

如果 `kill()` 失败，则不会发送任何信号。

RETURN VALUE

成功完成后，应返回 0。否则，应返回 -1 并设置 `errno` 以指示错误。

ERRORS

`kill()` 函数在以下情况下应失败：

- **[EINVAL]**

`sig` 参数的值是无效或不支持的信号编号。

- **[EPERM]**

进程没有权限向任何接收进程发送信号。

- **[ESRCH]**

找不到与 `pid` 指定的进程或进程组相对应的进程。

APPLICATION USAGE

无。

RATIONALE

`kill()` 的权限检查语义在 System V 和大多数其他实现（如 Version 7 或 4.3 BSD）之间有所不同。本卷 POSIX.1-2024 选择的语义与 System V 一致。具体来说，设置用户 ID 进程无法保护自己免受信号（或至少免受 SIGKILL）的影响，除非它更改其实际用户 ID。这个选择允许启动应用程序的用户即使应用程序更改了其有效用户 ID 也能向其发送信号。其他语义则赋予想要保护自己免受运行它的用户影响的应用程序更多权力。

一些实现在 `pid` 的绝对值大于某个最大值或其他特殊值时，为 `kill()` 函数提供语义扩展。负数值是 `kill()` 的标志。由于大多数实现在此情况下返回 [ESRCH]，因此此行为未包含在本卷 POSIX.1-2024 中，尽管符合标准的实现可以提供此类扩展。

不能向其发送信号的未指定进程可能包括调度器或 `init`。

最初有强烈意向规定，如果 `pid` 指定向调用进程发送信号且该信号未被阻塞，那么该信号应在 `kill()` 返回之前被传递。这将允许进程调用 `kill()` 并保证调用永不返回。然而，仅提供 `signal()` 函数的历史实现只满足本卷

POSIX.1-2024 中的较弱保证，因为它们只在进程每次进入内核时传递一个信号。对此类实现的修改以支持 `sigaction()` 函数通常需要在信号捕获函数返回后进入内核以恢复信号掩码。此类修改具有满足较强要求的效果，至少在使用 `sigaction()` 时是这样，但在使用 `signal()` 时不一定。标准开发者考虑过制定较强要求，但在使用 `signal()` 时除外，但认为这会不必要地复杂。鼓励实现者在可能的情况下满足较强要求。在实践中，较弱要求是相同的，除了在非常短的时间窗口内到达两个信号的罕见情况。此推理同样适用于 `sigprocmask()` 的类似要求。

在 4.2 BSD 中，SIGCONT 信号可以发送给任何后代进程，而无需进行用户 ID 安全检查。这允许作业控制 shell 继续作业，即使作业中的进程已更改其用户 ID (如 `su` 命令)。为了与会话概念的添加保持一致，通过允许向同一会话中的任何进程发送 SIGCONT 信号而无需进行用户 ID 安全检查，提供了类似功能。这比 BSD 限制更少，因为祖先进程 (在同一会话中) 现在可以成为接收者。这比 BSD 限制更多，因为形成新会话的后代进程现在需要接受用户 ID 检查。对于其他作业控制信号，不需要类似的安全放宽，因为这些信号通常由终端驱动程序发送以识别输入的特殊字符；终端驱动程序绕过所有安全检查。

在安全实现中，进程可能被限制向具有不同安全标签的进程发送信号。为了防止进程的存在或不存在被用作隐蔽通道，此类进程应对发送者显示不存在；也就是说，如果 `pid` 仅指此类进程，应返回 [ESRCH]，而不是 [EPERM]。

历史实现在 `pid` 指示僵尸进程的 `kill()` 结果上存在差异。一些实现在此类调用上指示成功 (取决于权限检查)，而其他实现则给出 [ESRCH] 错误。由于本卷 POSIX.1-2024 中进程生命周期的定义涵盖僵尸进程，因此描述的 [ESRCH] 错误在此情况下是不合适的，给出此错误的实现不符合标准。这意味着应用程序不能让父进程通过使用 `kill()` 向其发送空信号来检查特定子进程的终止，而必须使用 `waitpid()` 或 `waitid()`。

有人认为 `kill()` 的名称具有误导性，因为该函数并非总是旨在导致进程终止。然而，该名称在所有历史实现中都是通用的，任何更改都将与对现有应用程序代码进行最小更改的目标冲突。

FUTURE DIRECTIONS

无。

SEE ALSO

`getpid()` , `raise()` , `setsid()` , `sig2str()` , `sigaction()` ,
`sigqueue()` , `wait()`

CHANGE HISTORY

首次在 Issue 1 中发布。派生自 SVID 的 Issue 1。

Issue 5

DESCRIPTION 已更新以与 POSIX 线程扩展保持一致。

Issue 6

在 SYNOPSIS 中，移除了可选包含 `<sys/types.h>` 头文件。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 DESCRIPTION 中，第二段被重新措辞，以指示检查调用进程的已保存设置用户 ID，而不是其有效用户 ID。这是一个 FIPS 要求。
- 包含 `<sys/types.h>` 的要求已被移除。尽管 `<sys/types.h>` 对于先前 POSIX 规范的符合性实现是必需的，但对于 UNIX 应用程序来说不是必需的。
- 当 `pid` 为 -1 时的行为现在已指定。这在 POSIX.1-1988 标准中之前是明确未指定的。

规范性文本已更新，以避免对应用程序要求使用术语"必须"。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/51，更正了 RATIONALE 部分。

Issue 7

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0194 [765]。

Issue 8

应用了 Austin Group Defect 1138，将 `sig2str()` 添加到 SEE ALSO 部分。

1.80. labs, llabs — 返回长整型绝对值

SYNOPSIS (概要)

```
#include <stdlib.h>

long labs(long i);
long long llabs(long long i);
```

DESCRIPTION (描述)

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`labs()` 函数应计算 `long` 整型操作数 `i` 的绝对值。 `llabs()` 函数应计算 `long long` 整型操作数 `i` 的绝对值。如果结果无法表示，则行为未定义。

RETURN VALUE (返回值)

`labs()` 函数应返回 `long` 整型操作数的绝对值。

`llabs()` 函数应返回 `long long` 整型操作数的绝对值。

ERRORS (错误)

未定义错误。

以下部分为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

由于 POSIX.1 要求 `long` 和 `long long` 采用二进制补码表示，具有最大量级的负整数 `{LONG_MIN}` 和 `{LLONG_MIN}` 的绝对值无法表示，因此 `labs(LONG_MIN)` 和 `llabs(LLONG_MIN)` 是未定义的。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `abs()`
- XBD `<stdlib.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 4。源自 ISO C 标准。

Issue 6

添加 `llabs()` 函数以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

应用 SD5-XSH-ERN-152，更正了 RETURN VALUE 部分。

Issue 8

应用 Austin Group Defect 1108，更改了 APPLICATION USAGE 部分。

补充信息结束。

1.81. ldiv, lldiv — 计算长整型除法的商和余数

概要

```
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

这些函数应计算分子 `numer` 除以分母 `denom` 的商和余数。如果除法不精确，结果商是幅度更小且最接近日数商的 `long` 整数（对于 `ldiv()` 函数）或 `long long` 整数（对于 `lldiv()` 函数）。如果结果无法表示，则行为未定义；否则，`quot * denom + rem` 应等于 `numer`。

返回值

`ldiv()` 函数应返回 `ldiv_t` 类型的结构体，包含商和余数。该结构体应包含以下成员，顺序不限：

```
long    quot;    /* 商 */
long    rem;    /* 余数 */
```

`lldiv()` 函数应返回 `lldiv_t` 类型的结构体，包含商和余数。该结构体应包含以下成员，顺序不限：

```
long long    quot;    /* 商 */
long long    rem;    /* 余数 */
```

错误

未定义任何错误。

参见

- `div()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 4。源自 ISO C 标准。

Issue 6

添加 `lldiv()` 函数以与 ISO/IEC 9899:1999 标准保持一致。

1.82. llabs

概要

```
#include <stdlib.h>

long labs(long i);
long long llabs(long long i);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`labs()` 函数应计算 `long` 整型操作数 `i` 的绝对值。 `llabs()` 函数应计算 `long long` 整型操作数 `i` 的绝对值。如果结果无法表示，则行为是未定义的。

返回值

`labs()` 函数应返回 `long` 整型操作数的绝对值。

`llabs()` 函数应返回 `long long` 整型操作数的绝对值。

错误

未定义错误。

以下部分为参考信息。

示例

无。

应用程序用法

由于 POSIX.1 要求 `long` 和 `long long` 采用二进制补码表示，因此具有最大量级的负整数 `{LONG_MIN}` 和 `{LLONG_MIN}` 的绝对值是无法表示的，所以 `labs(LONG_MIN)` 和 `llabs(LLONG_MIN)` 是未定义的。

原理

无。

未来方向

无。

参见

- [abs\(\)](#)
- XBD

变更历史

首次发布于 Issue 4。源自 ISO C 标准。

Issue 6

为了与 ISO/IEC 9899:1999 标准保持一致，添加了 `llabs()` 函数。

Issue 7

应用了 SD5-XSH-ERN-152，修正了返回值部分。

Issue 8

应用了 Austin Group Defect 1108，更改了应用程序用法部分。

1.83. lldiv

SYNOPSIS

```
#include <stdlib.h>

ldiv_t ldiv(long numer, long denom);
lldiv_t lldiv(long long numer, long long denom);
```

DESCRIPTION

这些函数应计算分子 `numer` 除以分母 `denom` 的商和余数。如果除法不精确，结果商是幅度较小的最接近日数商的 `long` 整数（对于 `ldiv()` 函数）或 `long long` 整数（对于 `lldiv()` 函数）。如果结果无法表示，行为是未定义的；否则，`quot * denom + rem` 应等于 `numer`。

RETURN VALUE

`ldiv()` 函数应返回一个 `ldiv_t` 类型的结构体，包含商和余数。该结构体应按任意顺序包含以下成员：

```
long    quot;    /* 商 */
long    rem;    /* 余数 */
```

`lldiv()` 函数应返回一个 `lldiv_t` 类型的结构体，包含商和余数。该结构体应按任意顺序包含以下成员：

```
long long    quot;    /* 商 */
long long    rem;    /* 余数 */
```

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `div()`

XBD `<stdlib.h>`

CHANGE HISTORY

首次在 Issue 4 中发布。派生自 ISO C 标准。

Issue 6

为与 ISO/IEC 9899:1999 标准保持一致，添加了 `lldiv()` 函数。

1.84. `localeconv` — 返回本地化特定信息

概要

```
#include <locale.h>

struct lconv *localeconv(void);
```

描述

`localeconv()` 函数应当根据当前本地化环境的规则，用适于格式化数值（货币和非货币）数值来设置 `struct lconv` 类型对象的各个成员。

类型为 `char *` 的结构体成员是指向字符串的指针，其中任何一个（除了 `decimal_point`）都可以指向 "", 表示在当前本地化环境中该值不可用或长度为零。类型为 `char` 的成员是非负数，其中任何一个都可以是 {CHAR_MAX}，表示在当前本地化环境中该值不可用。

结构体成员

`char *decimal_point`

- 用于格式化非货币数值的基数字符。

`char *thousands_sep`

- 在格式化的非货币数值中，用于分隔小数点前数字组的字符。

`char *grouping`

- 一个字符串，其元素被视为单字节整数值，表示格式化的非货币数值中每个数字组的大小。

`char *int_curr_symbol`

- 适用于当前本地化环境的国际货币符号。前三个字符包含符合 ISO 4217:2015 标准的字母国际货币符号。第四个字符（紧邻空字节之前）是用于分隔国际货币符号和货币数值的字符。

`char *currency_symbol`

- 适用于当前本地化环境的本地货币符号。

`char *mon_decimal_point`

- 用于格式化货币数值的基数字符。

char *mon_thousands_sep

- 在格式化的货币数值中，小数点前数字组的分隔符。

char *mon_grouping

- 一个字符串，其元素被视为单字节整数值，表示格式化的货币数值中每个数字组的大小。

char *positive_sign

- 用于表示非负格式化货币数值的字符串。

char *negative_sign

- 用于表示负格式化货币数值的字符串。

char int_frac_digits

- 在国际格式化的货币数值中要显示的小数位数（小数点后的位数）。

char frac_digits

- 在格式化的货币数值中要显示的小数位数（小数点后的位数）。

char p_cs_precedes

- 如果 `currency_symbol` 位于非负格式化货币数值之前，设置为 1。如果符号位于数值之后，设置为 0。

char p_sep_by_space

- 设置为一个值，表示非负格式化货币数值的 `currency_symbol`、符号字符串和数值之间的分隔。

char n_cs_precedes

- 如果 `currency_symbol` 位于负格式化货币数值之前，设置为 1。如果符号位于数值之后，设置为 0。

char n_sep_by_space

- 设置为一个值，表示负格式化货币数值的 `currency_symbol`、符号字符串和数值之间的分隔。

char p_sign_posn

- 设置为一个值，表示非负格式化货币数值的 `positive_sign` 的位置。

char n_sign_posn

- 设置为一个值，表示负格式化货币数值的 `negative_sign` 的位置。

char int_p_cs_precedes

- 如果 `int_curr_symbol` 分别位于非负国际格式化货币数值之前或之后，设置为 1 或 0。

char int_n_cs_precedes

- 如果 `int_curr_symbol` 分别位于负国际格式化货币数值之前或之后，设置为 1 或 0。

char int_p_sep_by_space

- 设置为一个值，表示非负国际格式化货币数值的 `int_curr_symbol`、符号字符串和数值之间的分隔。

char int_n_sep_by_space

- 设置为一个值，表示负国际格式化货币数值的 `int_curr_symbol`、符号字符串和数值之间的分隔。

char int_p_sign_posn

- 设置为一个值，表示非负国际格式化货币数值的 `positive_sign` 的位置。

char int_n_sign_posn

- 设置为一个值，表示负国际格式化货币数值的 `negative_sign` 的位置。

分组解释

`grouping` 和 `mon_grouping` 的元素按照以下规则解释：

- {CHAR_MAX}**: 不再进行分组。
- 0**: 重复使用前一个元素来处理剩余的数字。
- 其他**: 整数值是组成当前组的数位数。检查下一个元素以确定当前组之前下一组数字的大小。

空格分隔值

`p_sep_by_space` 、 `n_sep_by_space` 、 `int_p_sep_by_space` 和 `int_n_sep_by_space` 的值按照以下规则解释：

- 0**: 货币符号和数值之间没有空格分隔。
- 1**: 如果货币符号和符号字符串相邻，则它们与数值之间用空格分隔；否则，货币符号与数值之间用空格分隔。

- 2: 如果货币符号和符号字符串相邻，则它们之间用空格分隔；否则，符号字符串与数值之间用空格分隔。

对于 `int_p_sep_by_space` 和 `int_n_sep_by_space`，使用 `int_curr_symbol` 的第四个字符代替空格。

符号位置值

`p_sign_posn`、`n_sign_posn`、`int_p_sign_posn` 和 `int_n_sign_posn` 的值按照以下规则解释：

- 0: 括号包围数值和 `currency_symbol` 或 `int_curr_symbol`。
- 1: 符号字符串位于数值和 `currency_symbol` 或 `int_curr_symbol` 之前。
- 2: 符号字符串位于数值和 `currency_symbol` 或 `int_curr_symbol` 之后。
- 3: 符号字符串紧邻 `currency_symbol` 或 `int_curr_symbol` 之前。
- 4: 符号字符串紧邻 `currency_symbol` 或 `int_curr_symbol` 之后。

实现应表现得像 POSIX.1-2024 卷中没有函数调用 `localeconv()`。

`localeconv()` 函数不需要是线程安全的；但是，`localeconv()` 应避免与所有其他函数发生数据竞争。

返回值

`localeconv()` 函数应当返回指向填充完成对象的指针。应用程序不应修改返回值指向的结构体，也不应修改结构体内指针指向的存储区域。返回的指针和结构体内的指针可能被后续的 `localeconv()` 调用无效化，或者结构体或存储区域可能被后续调用覆盖。此外，返回的指针和结构体内的指针可能被后续的 `setlocale()` 调用（使用 `LC_ALL`、`LC_MONETARY` 或 `LC_NUMERIC` 类别）或更改 `LC_MONETARY` 或 `LC_NUMERIC` 类别的 `use.locale()` 调用无效化，或者结构体或存储区域可能被覆盖。如果调用线程终止，返回的指针、结构体内的指针、结构体和存储区域也可能被无效化。

错误

没有定义错误。

示例

无。

应用程序用法

下表说明了四个国家可能用于格式化货币数值的规则：

国家	正值格式	负值格式	国际格式
意大利	€.1.230	-€.1.230	EUR.1.230
荷兰	€ 1.234,56	€ -1.234,56	EUR 1.234,56
挪威	kr1.234,56	kr1.234,56-	NOK 1.234,56
瑞士	SFr.1,234.56	SFr.1,234.56C	CHF 1,234.56

对于这四个国家，`localeconv()` 返回的结构体货币成员的相应值如下：

成员	意大利	荷兰	挪威	瑞士
<code>int_curr_symbol</code>	"EUR."	"EUR "	"NOK "	"CHF "
<code>currency_symbol</code>	"€."	"€"	"kr"	"SFr."
<code>mon_decimal_point</code>	"."	", "	", "	"."
<code>mon_thousands_sep</code>	"."	"."	"."	","
<code>mon_grouping</code>	"\3"	"\3"	"\3"	"\3"
<code>positive_sign</code>	""	""	""	""
<code>negative_sign</code>	"_"	"_"	"_"	"C"
<code>int_frac_digits</code>	0	2	2	2
<code>frac_digits</code>	0	2	2	2

成员	意大利	荷兰	挪威	瑞士
<code>p_cs_precedes</code>	1	1	1	1
<code>p_sep_by_space</code>	0	1	0	0
<code>n_cs_precedes</code>	1	1	1	1
<code>n_sep_by_space</code>	0	1	0	0
<code>p_sign_posn</code>	1	1	1	1
<code>n_sign_posn</code>	1	4	2	2
<code>int_p_cs_precedes</code>	1	1	1	1
<code>int_n_cs_precedes</code>	1	1	1	1
<code>int_p_sep_by_space</code>	0	0	0	0
<code>int_n_sep_by_space</code>	0	0	0	0
<code>int_p_sign_posn</code>	1	1	1	1
<code>int_n_sign_posn</code>	1	4	4	2

原理

无。

未来方向

无。

另见

- `fprintf()`
- `fscanf()`

- `isalpha()`
- `nl_langinfo()`
- `setlocale()`
- `strcat()`
- `strchr()`
- `strcmp()`
- `strcoll()`
- `strcpy()`
- `strftime()`
- `strlen()`
- `strpbrk()`
- `strspn()`
- `strtok()`
- `strxfrm()`
- `strtod()`
- `use locale()`
- XBD `<langinfo.h>`, `<locale.h>`

变更历史

首次发布于 Issue 4。源自 ANSI C 标准。

Issue 6

- 在描述中添加了说明此函数不需要可重入的注释。
- 重写返回值部分以避免使用"必须"一词。
- 更新此参考页面以与 ISO/IEC 9899:1999 标准对齐。
- 纳入 ISO/IEC 9899:1999 标准、技术勘误表 1。
- 应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/31，删除对 `int_curr_symbol` 的引用并更新 `p_sep_by_space` 和 `n_sep_by_space` 的描述。这些更改是为了与 ISO C 标准对齐。

Issue 7

- 应用 Austin Group 解释 1003.1-2001 #156。
- 更新 `int_curr_symbol` 和 `currency_symbol` 的定义。
- 更新应用程序用法部分的示例。
- 应用 POSIX.1-2008, 技术勘误表 1, XSH/TC1-2008/0362 [75]。
- 应用 POSIX.1-2008, 技术勘误表 2, XSH/TC2-2008/0200 [656]。

Issue 8

- 应用 Austin Group 缺陷 1302, 使此函数与 ISO/IEC 9899:2018 标准对齐。

1.85. `localtime`, `localtime_r` — 将时间值转换为本地时间的分解表示

概要

```
#include <time.h>

struct tm *localtime(const time_t *timer);

[CX] struct tm *localtime_r(const time_t *restrict timer,
                           struct tm *restrict result);
```

描述

对于 `localtime()`：

[CX] 本参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 版本遵循 ISO C 标准。

`localtime()` 函数应将 `timer` 指向的自纪元 (Epoch) 以来的秒数时间转换为分解时间，以本地时间表示。该函数应校正时区和任何季节性时间调整。[CX] 本地时区信息应设置为 `localtime()` 调用 `tzset()` 的方式。

作为 `localtime()` 参数使用的自纪元以来的秒数时间与 `tm` 结构（在 `<time.h>` 头文件中定义）之间的关系是，结果应按照自纪元以来秒数定义中给出的表达式指定（参见XBD 4.19 自纪元以来的秒数），并进行时区和任何季节性时间调整的校正，其中结构和表达式中的名称相互对应。

同样的关系应适用于 `localtime_r()`。

`localtime()` 函数不需要是线程安全的；然而，`localtime()` 应避免与除自身、`asctime()`、`ctime()` 和 `gmtime()` 之外的所有函数发生数据竞争。

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解时间结构和类型为 `char` 的数组。执行任何返回指向这些对象类型之一的指针的函数都可能覆盖从先前对其中任何函数的调用返回的值所指向的任何相同类型对象中的信息。

[CX] `localtime_r()` 函数应将 `timer` 指向的自纪元以来的秒数时间转换为存储在 `result` 所指向结构中的分解时间。`localtime_r()` 函数还应返回指向同一结构的指针。

与 `localtime()` 不同，`localtime_r()` 函数不需要设置 `tzname`。如果 `localtime_r()` 设置了 `tzname`，它还应设置 `daylight` 和 `timezone`。如果 `localtime_r()` 没有设置 `tzname`，则不应设置 `daylight`，也不应设置 `timezone`。如果在 `TZ` 值随后被修改后访问 `tm` 结构成员 `tm_zone`，则行为未定义。

返回值

成功完成后，`localtime()` 函数应返回指向分解时间结构的指针。如果检测到错误，`localtime()` 应返回空指针[CX]并设置 `errno` 以指示错误。

成功完成后，`localtime_r()` 应返回指向参数 `result` 所指向结构的指针。如果检测到错误，`localtime_r()` 应返回空指针并设置 `errno` 以指示错误。

错误

`localtime()` [CX] 和 `localtime_r()` 函数可能在以下情况下失败：

- **[EOVERFLOW]**: [CX] 结果无法表示。

示例

获取本地日期和时间

以下示例使用 `time()` 函数计算自1970年1月1日0:00 UTC（纪元）以来经过的时间（秒），使用 `localtime()` 将该值转换为分解时间，并使用 `asctime()` 将分解时间值转换为可打印字符串。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t result;

    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
    return(0);
}
```

此示例以如下形式将当前时间写入 `stdout`：

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

获取文件的修改时间

以下示例在本地时区中打印给定文件的最后数据修改时间戳。

```
#include <stdio.h>
#include <time.h>
#include <sys/stat.h>

int
print_file_time(const char *pathname)
{
    struct stat statbuf;
    struct tm *tm;
    char timestr[BUFSIZ];

    if(stat(pathname, &statbuf) == -1)
        return -1;
    if((tm = localtime(&statbuf.st_mtime)) == NULL)
        return -1;
    if(strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S",
                tm) == NULL)
        return -1;
    printf("%s: %s.%09ld\n", pathname, timestr, statbuf.st_mtim);
    return 0;
}
```

为事件计时

以下示例获取当前时间，使用 `localtime()` 和 `asctime()` 将其转换为字符串，并使用 `fputs()` 将其打印到标准输出。然后打印到正计时事件的分钟数。

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
```

```
minutes_to_event);  
...
```

应用程序用法

`localtime_r()` 函数是线程安全的，并且在用户提供的缓冲区中返回值，而不是使用可能被每次调用覆盖的静态数据区域。

原理

无。

未来方向

无。

另请参见

`asctime()` , `clock()` , `ctime()` , `difftime()` , `futimens()` ,
`getdate()` , `gmtime()` , `mktime()` , `strftime()` , `strptime()` ,
`time()` , `tzset()`

XBD 4.19 自纪元以来的秒数, `<time.h>`

变更历史

首次在Issue 1中发布。源自SVID的Issue 1。

Issue 5

添加了一个说明，指出 `localtime()` 函数不需要是可重入的。

`localtime_r()` 函数被包含以与POSIX线程扩展保持一致。

Issue 6

`localtime_r()` 函数被标记为线程安全函数选项的一部分。

超出ISO C标准的扩展被标记。

应用程序用法部分被更新，包含关于线程安全函数及其避免可能使用静态数据区域的说明。

`restrict` 关键字被添加到 `localtime_r()` 原型以与ISO/IEC 9899:1999标准保持一致。

添加了示例。

应用了IEEE Std 1003.1-2001/Cor 1-2002，项目XSH/TC1/D6/32，添加了[EOVERFLOW]错误。

应用了IEEE Std 1003.1-2001/Cor 2-2004，项目XSH/TC2/D6/55，更新了 `localtime_r()` 的错误处理。

应用了IEEE Std 1003.1-2001/Cor 2-2004，项目XSH/TC2/D6/56，添加了一个要求：如果 `localtime_r()` 没有设置 `tzname` 变量，则不应设置 `daylight` 或 `timezone` 变量。在支持XSI的系统上，`daylight`、`timezone` 和 `tzname` 变量都应该被设置为提供相同时区的信息。这更新了 `localtime_r()` 的描述以提及 `daylight` 和 `timezone` 以及 `tzname`。另请参见部分被更新。

Issue 7

应用了Austin Group解释1003.1-2001 #156。

`localtime_r()` 函数从线程安全函数选项移动到基础。

对示例部分进行了与支持细粒度时间戳相关的更改。

应用了POSIX.1-2008，技术勘误1，XSH/TC1-2008/0363 [291]。

应用了POSIX.1-2008，技术勘误2，XSH/TC2-2008/0201 [664]。

Issue 8

应用了Austin Group缺陷1125，将"使用本地时区信息"更改为"应设置本地时区信息"。

应用了Austin Group缺陷1302，使 `localtime()` 函数与ISO/IEC 9899:2018标准保持一致。

应用了Austin Group缺陷1376，从源自ISO C标准的一些文本中移除CX阴影并将其更新以匹配ISO C标准。

应用了Austin Group缺陷1533，向 `tm` 结构添加了 `tm_gmtoff` 和 `tm_zone`。

应用了Austin Group缺陷1570，移除了"=="中的额外空格。

1.86. `localtime`, `localtime_r` - 将时间值转换为分解的本地时间

概要

```
#include <time.h>

struct tm *localtime(const time_t *timer);

[CX] struct tm *localtime_r(const time_t *restrict timer,
                           struct tm *restrict result);
```

描述

对于 `localtime()`：

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`localtime()` 函数应将自纪元以来以秒为单位的时间（由 `timer` 指向）转换为分解的时间，表示为本地时间。该函数会校正时区和任何季节性时间调整。[CX] 本地时区信息的设置应如同 `localtime()` 调用 `tzset()` 一样。

作为 `localtime()` 参数使用的自纪元以来的时间（以秒为单位）与 `tm` 结构体（在 `<time.h>` 头文件中定义）之间的关系是：结果应按照自纪元以来的秒数的定义中给出的表达式指定（见 XBD 4.19 自纪元以来的秒数），校正时区和任何季节性时间调整，其中结构体和表达式中的名称相对应。

同样的关系适用于 `localtime_r()`。

`localtime()` 函数不需要是线程安全的；但是，`localtime()` 应避免与除自身、`asctime()`、`ctime()` 和 `gmtime()` 之外的所有函数发生数据竞争。

`asctime()`、`ctime()`、`gmtime()` 和 `localtime()` 函数应在两个静态对象之一中返回值：一个分解的时间结构体和一个 `char` 类型的数组。执行返回指向这些对象类型之一的指针的任何函数可能会覆盖任何先前调用它们中任何一个所返回的值指向的相同类型的任何对象中的信息。

[CX] `localtime_r()` 函数应将自纪元以来以秒为单位的时间（由 `timer` 指向）转换为存储在 `result` 指向的结构体中的分解时间。`localtime_r()` 函数还应返回指向同一结构体的指针。

与 `localtime()` 不同，`localtime_r()` 函数不需要设置 `tzname`。如果 `localtime_r()` 设置了 `tzname`，它还应设置 `daylight` 和 `timezone`。如果 `localtime_r()` 没有设置 `tzname`，它不应设置 `daylight` 也不应设置 `timezone`。如果在 `TZ` 的值随后被修改后访问 `tm` 结构体成员 `tm_zone`，则行为未定义。

返回值

成功完成时，`localtime()` 函数应返回指向分解时间结构体的指针。如果检测到错误，`localtime()` 应返回空指针 [CX] 并设置 `errno` 以指示错误。

成功完成时，`localtime_r()` 应返回指向参数 `result` 指向的结构体的指针。如果检测到错误，`localtime_r()` 应返回空指针并设置 `errno` 以指示错误。

错误

`localtime()` [CX] 和 `localtime_r()` 函数在以下情况下可能失败：

- **EOVERFLOW** [CX] 结果无法表示。

示例

获取本地日期和时间

以下示例使用 `time()` 函数计算自 1970 年 1 月 1 日 0:00 UTC（纪元）以来经过的时间（以秒为单位），使用 `localtime()` 将该值转换为分解的时间，并使用 `asctime()` 将分解的时间值转换为可打印的字符串。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t result;

    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
```

```
    return(0);
}
```

此示例将当前时间写入 `stdout`，格式如下：

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

获取文件的修改时间

以下示例在本地时区中打印给定文件的最后数据修改时间戳。

```
#include <stdio.h>
#include <time.h>
#include <sys/stat.h>

int
print_file_time(const char *pathname)
{
    struct stat statbuf;
    struct tm *tm;
    char timestr[BUFSIZ];

    if(stat(pathname, &statbuf) == -1)
        return -1;
    if((tm = localtime(&statbuf.st_mtime)) == NULL)
        return -1;
    if(strftime(timestr, sizeof(timestr), "%Y-%m-%d %H:%M:%S",
                tm) == -1)
        return -1;
    printf("%s: %s.%09ld\n", pathname, timestr, statbuf.st_mtim
    return 0;
}
```

事件计时

以下示例获取当前时间，使用 `localtime()` 和 `asctime()` 将其转换为字符串，并使用 `fputs()` 将其打印到标准输出。然后打印到正在计时的事件的分钟数。

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
```

```
...
time(&now);
printf("The time is ");
fputs(asctime(localtime(&now)), stdout);
printf("There are still %d minutes to the event.\n",
       minutes_to_event);
...
...
```

应用程序使用

`localtime_r()` 函数是线程安全的，并在用户提供的缓冲区中返回值，而不是可能使用可能被每次调用覆盖的静态数据区域。

基本原理

无。

未来方向

无。

参见

- `asctime()`
- `clock()`
- `ctime()`
- `difftime()`
- `futimens()`
- `getdate()`
- `gmtime()`
- `mktime()`
- `strftime()`
- `strptime()`
- `time()`

- `tzset()`

XBD 4.19 自纪元以来的秒数, `<time.h>`

更改历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

在描述中添加了一个注释, 表明 `localtime()` 函数不需要是可重入的。

`localtime_r()` 函数被包含用于与 POSIX 线程扩展对齐。

Issue 6

`localtime_r()` 函数被标记为线程安全函数选项的一部分。

超过 ISO C 标准的扩展被标记。

应用程序使用部分被更新, 包含有关线程安全函数及其避免可能使用静态数据区域的注释。

`restrict` 关键字被添加到 `localtime_r()` 原型中, 以与 ISO/IEC 9899:1999 标准对齐。

添加了示例。

应用 IEEE Std 1003.1-2001/Cor 1-2002, 项目 XSH/TC1/D6/32, 添加了 [EOVERFLOW] 错误。

应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/55, 更新了 `localtime_r()` 的错误处理。

应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/56, 添加了一个要求: 如果 `localtime_r()` 不设置 `tzname` 变量, 则不应设置 `daylight` 或 `timezone` 变量。在支持 XSI 的系统上, `daylight`、`timezone` 和 `tzname` 变量都应设置为提供相同时区的信息。这更新了 `localtime_r()` 的描述, 提及 `daylight` 和 `timezone` 以及 `tzname`。更新了参见部分。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #156。

`localtime_r()` 函数从线程安全函数选项移至基础。

对示例部分进行了与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0363 [291]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0201 [664]。

Issue 8

应用 Austin Group Defect 1125，将"使用本地时区信息"更改为"应设置本地时区信息"。

应用 Austin Group Defect 1302，使 `localtime()` 函数与 ISO/IEC 9899:2018 标准对齐。

应用 Austin Group Defect 1376，从源自 ISO C 标准的某些文本中移除 CX 着色，并更新它以匹配 ISO C 标准。

应用 Austin Group Defect 1533，向 `tm` 结构体添加 `tm_gmtoff` 和 `tm_zone`。

应用 Austin Group Defect 1570，移除"=="中的额外间距。

1.87. longjmp — 非本地跳转

概要

```
#include <setjmp.h>

_Noreturn void longjmp(jmp_buf env, int val);
```

描述

`longjmp()` 函数应当恢复在同一进程中由最近的 `setjmp()` 调用所保存的环境，该调用使用相应的 `jmp_buf` 参数。如果具有相应 `jmp_buf` 的最近 `setjmp()` 调用发生在另一个线程中，或者不存在这样的调用，或者包含 `setjmp()` 调用的函数在此期间已经终止执行，或者 `setjmp()` 调用在具有可变修改类型的标识符作用域内，而执行在此期间已经离开该作用域，则行为是未定义的。`longjmp()` 是否恢复信号掩码、保持信号掩码不变，或者将其恢复到 `setjmp()` 调用时的值是未指定的。

所有可访问对象都具有值，抽象机的所有其他组件都处于状态（例如，浮点状态标志和打开的文件），这些值和状态都是 `longjmp()` 被调用时的状态，除非满足以下所有条件的自动存储期限对象的值是未指定的：

- 它们是包含相应 `setjmp()` 调用的函数的局部变量。
- 它们不具有 `volatile` 限定类型。
- 它们在 `setjmp()` 调用和 `longjmp()` 调用之间被修改。

虽然 `longjmp()` 是一个异步信号安全函数，如果它从信号处理程序中被调用，而该信号处理程序中断了一个非异步信号安全函数或等效函数（如在从 `main()` 的初始调用返回后执行的相当于 `exit()` 的处理），则任何后续对非异步信号安全函数或等效函数调用的行为是未定义的。

在调用线程中未执行 `jmp_buf` 结构初始化的情况下调用 `longjmp()` 的效果是未定义的。

返回值

`longjmp()` 完成后，线程执行应当继续，就像相应的 `setjmp()` 调用刚刚返回了由 `val` 指定的值一样。`longjmp()` 函数不应导致 `setjmp()` 返回 0；

如果 `val` 为 0, `setjmp()` 应当返回 1。

错误

未定义错误。

应用程序用法

其行为依赖于信号掩码值的应用程序不应使用 `longjmp()` 和 `setjmp()`，因为它们对信号掩码的影响是未指定的，而应使用 `siglongjmp()` 和 `sigsetjmp()` 函数（这些函数可以在应用程序控制下保存和恢复信号掩码）。

建议应用程序不要从信号处理程序中调用 `longjmp()` 或 `siglongjmp()`。为了避免从信号处理程序中调用这些函数时的未定义行为，应用程序需要确保以下两件事之一：

1. 在调用 `longjmp()` 或 `siglongjmp()` 后，进程只调用异步信号安全函数，并且不从 `main()` 的初始调用返回。
2. 任何处理程序调用 `longjmp()` 或 `siglongjmp()` 的信号在每次对非异步信号安全函数的调用期间都被阻塞，并且在从 `main()` 的初始调用返回后不进行此类调用。

参见

- `setjmp()`
- `sigaction()`
- `siglongjmp()`
- `sigsetjmp()`
- `<setjmp.h>`

变更历史

首次发布于 Issue 1

派生自 SVID 的 Issue 1。

Issue 5

描述已更新，以与 POSIX 线程扩展保持一致。

Issue 6

超出 ISO C 标准的扩展被标记。

以下对 POSIX 实现的新要求来源于与单一 UNIX 规范的对齐：

- 描述现在明确地将 `longjmp()` 对信号掩码的影响设为未指定。

描述已更新，以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0365 [394]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0202 [516]。

Issue 8

应用了 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准保持一致。

1.88. malloc — 内存分配器

概要

```
#include <stdlib.h>

void *malloc(size_t size);
```

描述

`malloc()` 函数应为指定字节大小为 `size` 的对象分配未使用的空间，该对象的值是未指定的。

通过连续调用 `malloc()` 分配的存储的顺序和连续性是未指定的。如果分配成功，返回的指针应适当对齐，以便可以将其赋给具有基本对齐要求的任何类型对象的指针，然后用于在分配的空间中访问此类对象（直到空间被显式释放或重新分配）。每次此类分配都会产生一个与任何其他对象不重叠的对象指针。返回的指针指向分配空间的开头（最低字节地址）。如果无法分配空间，应返回空指针。如果请求的空间大小为 0，其行为是实现定义的：要么返回空指针，要么行为如同大小为某个非零值，除非如果使用返回的指针访问对象，则行为是未定义的。

为确定数据竞争的存在，`malloc()` 的行为应如同它只访问可通过其参数访问的内存位置，而不访问其他静态持续时间存储。但是，该函数可以显式修改其分配的存储。调用 `aligned_alloc()`、`calloc()`、`free()`、`malloc()`、`posix_memalign()`、`reallocarray()` 和 `realloc()` 来分配或释放特定内存区域的调用应按单一总顺序发生（参见 [4.15.1 内存排序](#)），并且每个此类释放调用应与此顺序中的下一次分配（如果有）同步。

返回值

成功完成时，`malloc()` 应返回指向分配空间的指针；如果 `size` 为 0，应用程序应确保不使用该指针访问对象。

否则，它应返回空指针并设置 `errno` 以指示错误。

错误

`malloc()` 函数在以下情况下应失败：

- [ENOMEM] 可用存储空间不足。

`malloc()` 函数在以下情况下可能失败：

- [EINVAL] `size` 为 0 且实现不支持大小为 0 的分配。

示例

无。

应用程序用法

无。

原理说明

某些实现将 `errno` 设置为 [EAGAIN] 以表示如果重试可能成功的内存分配失败，将 `errno` 设置为 [ENOMEM] 表示由于配置限制等原因不太可能成功的失败。[2.3 错误编号](#) 允许这种行为；当多个错误条件同时为真时，它们之间没有优先级。

未来方向

无。

另请参阅

- `aligned_alloc()`
- `calloc()`
- `free()`
- `getrlimit()`
- `posix_memalign()`

- `realloc()`

XBD `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 部分，添加了设置 `errno` 以指示错误的要求。
- 添加了 [ENOMEM] 错误条件。

Issue 7

应用了 POSIX.1-2008 技术勘误 2，XSH/TC2-2008/0203 [526]。

Issue 8

应用了 Austin Group 缺陷 374，更改了与大小为 0 的分配相关的 RETURN VALUE 和 ERRORS 部分。

应用了 Austin Group 缺陷 1302，使此函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group 缺陷 1387 和 1489，更改了 RATIONALE 部分。

1.89. memchr

概要

```
#include <string.h>

void *memchr(const void *s, int c, size_t n);
```

描述

`memchr()` 函数应在 `s` 指向的初始 `n` 个字节（每个字节都解释为 `unsigned char`）中定位 `c`（转换为 `unsigned char`）的首次出现。

实现在行为上应表现为顺序读取字节，并在找到匹配字节后立即停止。

对于有效输入，`memchr()` 函数不应改变 `errno` 的设置。

返回值

`memchr()` 函数应返回指向所定位字节的指针，如果未找到该字节，则返回空指针。

错误

未定义错误。

示例

无。

应用用法

无。

基本原理

无。

未来方向

无.

另请参阅

`<string.h>`

更改历史

首次发布于 Issue 1。派生自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0374 [110]。

Issue 8

应用了 Austin Group Defect 448，增加了 `memchr()` 对于有效输入不改变 `errno` 设置的要求。

应用了 Austin Group Defect 1302，使此函数与 ISO/IEC 9899:2018 标准保持一致。

1.90. memcmp — 内存字节比较

SYNOPSIS

```
#include <string.h>

int memcmp(const void *s1, const void *s2, size_t n);
```

DESCRIPTION

`memcmp()` 函数应将 `s1` 所指向对象的前 `n` 个字节（每个字节都被解释为 `unsigned char` 类型）与 `s2` 所指向对象的前 `n` 个字节进行比较。

非零返回值的符号应由所比较对象中第一对不同的字节（都被解释为 `unsigned char` 类型）的值之间的差的符号确定。

`memcmp()` 函数在有效输入上不应改变 `errno` 的设置。

RETURN VALUE

如果 `s1` 所指向的对象大于、等于或小于 `s2` 所指向的对象，`memcmp()` 函数应分别返回大于、等于或小于 0 的整数。

ERRORS

未定义错误。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `<string.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 448，增加了 `memcmp()` 在有效输入上不改变 `errno` 设置的要求。

1.91. memcpy — 内存字节复制

概要

```
#include <string.h>

void *memcpy(void *restrict s1, const void *restrict s2, size_t
```



描述

[CX] 本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

`memcpy()` 函数应将 `n` 个字节从 `s2` 指向的对象复制到 `s1` 指向的对象中。如果在重叠的对象之间进行复制，则行为未定义。

[CX] `memcpy()` 函数在有效输入时不应改变 `errno` 的设置。

返回值

`memcpy()` 函数应返回 `s1`；没有保留返回值来指示错误。

错误

未定义任何错误。

以下各节为信息性内容。

示例

无。

应用用法

`memcpy()` 函数不检查接收内存区域的溢出。

基本原理

无。

未来方向

无。

参见

XBD `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

`memcpy()` 原型已更新，以与 ISO/IEC 9899:1999 标准对齐。

Issue 8

应用了 Austin Group Defect 448，增加了 `memcpy()` 在有效输入时不改变 `errno` 设置的要求。

1.92. memmove

SYNOPSIS

```
#include <string.h>

void *memmove(void *s1, const void *s2, size_t n);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 遵从 ISO C 标准。

`memmove()` 函数应将 `n` 个字节从 `s2` 指向的对象复制到 `s1` 指向的对象中。复制过程的行为如同：先将 `s2` 指向的对象中的 `n` 个字节复制到一个与 `s1` 和 `s2` 指向的对象都不重叠的、包含 `n` 个字节的临时数组中，然后再将临时数组中的 `n` 个字节复制到 `s1` 指向的对象中。

[CX] `memmove()` 函数在有效输入时不应更改 `errno` 的设置。

RETURN VALUE

`memmove()` 函数应返回 `s1`；没有保留返回值来指示错误。

ERRORS

未定义错误。

以下各节为资料性内容。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

XBD `<string.h>`

CHANGE HISTORY

首次发布于 Issue 4。源自 ANSI C 标准。

Issue 8

应用了 Austin Group 缺陷 448，添加了 `memmove()` 在有效输入时不应更改 `errno` 设置的要求。

1.93. memset

SYNOPSIS

```
#include <string.h>

void *memset(void *s, int c, size_t n);
```

DESCRIPTION

[选项开始] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵从 ISO C 标准。[选项结束]

`memset()` 函数应将 `c` (转换为 `unsigned char` 类型) 复制到由 `s` 指向的对象的前 `n` 个字节中的每一个字节。

[选项开始] `memset()` 函数在有效输入时不应更改 `errno` 的设置。[选项结束]

RETURN VALUE

`memset()` 函数应返回 `s`；没有保留用于指示错误的返回值。

ERRORS

未定义错误。

以下章节为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

XBD \<string.h>

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 448，增加了 `memset()` 在有效输入时不更改 `errno` 设置的要求。

信息性文本结束。

1.94. mktimes — 将分解时间转换为自纪元以来的时间

概要

```
#include <time.h>

time_t mktimes(struct tm *timeptr);
```

描述

`mktimes()` 函数应将由 `timeptr` 指向的结构体的某些成员所表示的分解时间（以本地时间表示）转换为与 `time()` 函数返回值具有相同编码的自纪元以来的时间值。`mktimes()` 函数应仅使用由 `timeptr` 指向的结构体的 `tm_year`、`tm_mon`、`tm_mday`、`tm_hour`、`tm_min`、`tm_sec` 和 `tm_isdst` 成员；这些成员的值不应被限制在 `<time.h>` 中描述的范围内。

本地时区信息应设置为如同 `mktimes()` 调用了 `tzset()` 一样。

`mktimes()` 函数应计算自纪元以来的秒数时间以返回，该计算应按照以下步骤通过操作 `tm` 结构体的成员来实现：

1. `tm_sec` 成员可以但不应被调整到 0 到 60（含）的范围内。对于向 `tm_sec` 添加或减去的每 60 秒，应分别保存 1 分钟的减少或增加，以供后续应用。
2. `tm_min` 成员应被调整到 0 到 59（含）的范围内，然后应用任何保存的分钟减少或增加，如有必要，重复范围调整。对于向 `tm_min` 添加或减去的每 60 分钟，应分别保存 1 小时的减少或增加，以供后续应用。
3. `tm_hour` 成员应被调整到 0 到 23（含）的范围内，然后应用任何保存的小时减少或增加，如有必要，重复范围调整。对于向 `tm_hour` 添加或减去的每 24 小时，应分别保存 1 天的减少或增加，以供后续应用。
4. `tm_mon` 成员应被调整到 0 到 11（含）的范围内。对于向 `tm_mon` 添加或减去的每 12 个月，应分别保存 1 年的减少或增加，以供后续使用。
5. `tm_mday` 成员应被调整到 1 到 31（含）的范围内，然后应用任何保存的天数减少或增加，如有必要，重复范围调整。向下调整应通过从 `tm_mon` +1 月份（该年通过将任何保存的年增加 / 减少加到 `tm_year` +1900 的值得到）的天数（根据公历）中减去来应用，然后将 `tm_mon` 增加 1，根据需要重复。向上调整应通过向 `tm_mon` +1 月份之前

月份（该年通过将任何保存的年增加/减少加到 `tm_year` +1900 的值得到）的天数中添加来应用，然后将 `tm_mon` 减少 1，根据需要重复。在这些调整期间，如有必要，应应用步骤 4 将 `tm_mon` 值保持在 0 到 11（含）的范围内。

6. 如果 `tm_mday` 成员大于 `tm_mon` +1 月份（该年通过将任何保存的年增加/减少加到 `tm_year` +1900 的值得到）的天数，应从 `tm_mday` 中减去该天数，并将 `tm_mon` 增加 1。如果这导致 `tm_mon` 的值为 12，应应用步骤 4。
7. 应根据相关 `tm` 结构体成员的范围修正值（或成员未范围修正的原始值）计算自协调世界时纪元以来的秒数，如自纪元以来秒数的定义中给出的表达式所指定（参见 XBD 4.19 自纪元以来的秒数），其中结构和表达式中的名称（除 `tm_year` 和 `tm_yday` 外）相对应，表达式中使用的 `tm_year` 值是结构体中的 `tm_year` 加上/减去任何保存的年增加/减少，表达式中使用的 `tm_yday` 值是该年的从 0 到 365（含）的年份中的第几天，从 `tm` 结构体的 `tm_mon` 和 `tm_mday` 成员计算得出。
8. 应针对本地时区标准时间与协调世界时的偏移量修正自纪元以来的时间。
9. 应进一步修正自纪元以来的时间（如适用——见下文）夏令时。

如果时区是包括夏令时（DST）调整的时区，`tm` 结构体中 `tm_isdst` 的值控制 `mkttime()` 是否通过 DST 偏移量调整计算的自纪元以来的秒数值（在进行时区调整之后），如下所示：

- 如果 `tm_isdst` 为零，`mkttime()` 不应通过 DST 偏移量进一步调整自纪元以来的秒数。
- 如果 `tm_isdst` 为正数，`mkttime()` 应通过 DST 偏移量进一步调整自纪元以来的秒数。
- 如果 `tm_isdst` 为负数，`mkttime()` 应尝试确定指定时间是否 DST 生效；如果确定 DST 生效，它应产生与等效调用具有正 `tm_isdst` 值相同的结果，否则应产生与等效调用 `tm_isdst` 值为零相同的结果。如果分解时间指定的时间在 DST 转换时被跳过或重复，则未指定 `mkttime()` 是产生与等效调用具有正 `tm_isdst` 值相同的结果，还是产生与等效调用 `tm_isdst` 值为零相同的结果。

如果 `TZ` 环境变量指定了一个地理时区，该时区的实现时区数据库包含该时区标准时间与协调世界时偏移量的历史或未来更改，并且分解时间对应于由于此类更改发生而被（或将被）跳过或重复的时间，`mkttime()` 应使用更改前生效的偏移量或更改后生效的偏移量计算自纪元以来的时间值。

成功完成后，结构体的成员应设置为通过调用 `localtime()` 并以计算的自纪元以来的时间作为其参数将返回的值。

返回值

`mktime()` 函数应返回计算的自纪元以来的时间，编码为 `time_t` 类型的值。如果自纪元以来的时间不能表示为 `time_t` 或要在 `timeptr` 指向的结构体的 `tm_year` 成员中返回的值不能表示为 `int`，函数应返回值 `(time_t)-1` 并设置 `errno` 为 `[EOVERFLOW]`，并且不应更改结构体的 `tm_wday` 组件的值。

由于 `(time_t)-1` 是成功调用 `mktime()` 的有效返回值，希望检查错误情况的应用程序应在调用 `mktime()` 之前将 `tm_wday` 设置为小于 0 或大于 6 的值。返回时，如果 `tm_wday` 未更改，则发生了错误。

错误

`mktime()` 函数可能在以下情况下失败：

`[EOVERFLOW]`

结果无法表示。

示例

2001年7月4日是星期几？

```
#include <stdio.h>
#include <time.h>

struct tm time_str;

char daybuf[20];

int main(void)
{
    time_str.tm_year = 2001 - 1900;
    time_str.tm_mon = 7 - 1;
    time_str.tm_mday = 4;
    time_str.tm_hour = 0;
    time_str.tm_min = 0;
    time_str.tm_sec = 1;
    time_str.tm_isdst = -1;
    time_str.tm_wday = -1;
```

```
if (mktime(&time_str) == (time_t)-1 && time_str.tm_wday ==  
    (void)puts("-未知-");  
else {  
    (void)strftime(daybuf, sizeof(daybuf), "%A", &time_str)  
    (void)puts(daybuf);  
}  
return 0;  
}
```

应用程序用法

当使用 `mktime()` 向本地时区的分解时间添加或减去固定时间段（总是对应固定秒数的时间段）时，对于任意的 `TZ`，只有通过使用 `mktime()` 将原始分解时间转换为自纪元以来的时间，向该值添加或减去所需秒数，然后使用结果调用 `localtime()`，才能确保可靠的结果。在调用 `mktime()` 之前调整分解时间的替代方法，如果原始时间和更新时间位于地理时区更改的不同侧，可能会产生意外结果。在遵循不范围修正 `tm_sec` 建议的实现上（参见 DESCRIPTION 中的步骤1），也可以通过向 `tm_sec` 添加或减去所需秒数（而不修改 `tm` 结构体的任何其他成员）来确保可靠的结果。在需要移植到 `time_t` 编码不是秒数的非 POSIX 系统的应用程序中，建议使用条件编译，以便在可能时在 `mktime()` 返回值上执行调整，否则在 `tm_sec` 成员上执行调整。对于已知没有地理时区更改的时区，如 `TZ=UTC0`，仅使用 `mktime()` 的调整不存在此问题。

当同时进行多个调整时，`mktime()` 函数解释超出范围的 `tm` 结构体字段的方式可能不会产生预期结果。例如，如果应用程序尝试通过调用 `localtime()` 先后退一天然后再后退一年，减少 `tm_mday` 和 `tm_year`，然后调用 `mktime()`，如果在 2021-03-01 上调用，这不会产生预期结果，因为 `mktime()` 会看到提供的年份为 2020（闰年）并将 Mar 0 修正为 Feb 29，而预期的结果是 Feb 28。通过在调整顺序重要时一次执行多个调整，可以避免此类问题。

`mktime()` 处理某些调整的示例如下：

- 如果在非闰年中给出 Feb 29，它将其视为 Feb 28 之后的日期并返回 Mar 1。
- 如果给出 Feb 0，它将其视为 Feb 1 之前的日期并返回 Jan 31。
- 如果给出 21:65，它将其视为 21:59 之后的 6 分钟并返回 22:05。
- 如果在 DST 生效的时间给出 `tm_isdst` =0，它返回正的 `tm_isdst` 并适当地更改其他字段。

- 如果存在 DST 转换，其中 02:00 标准时间变为 03:00 DST，并且 `mktimed()` 获得 02:30（带有负 `tm_isdst`），它将其视为 02:00 标准时之后的 30 分钟或 03:00 DST 之前的 30 分钟，并分别返回零或正的 `tm_isdst`，并适当地更改 `tm_hour` 字段。
- 如果地理时区更改其 UTC 偏移量，使得"旧 00:00"变为"新 00:30"，并且 `mktimed()` 获得 00:20，它将其视为"旧 00:00"之后的 20 分钟或"新 00:30"之前的 10 分钟，并返回适当更改的 `struct tm` 字段。

如果应用程序想要检查给定的分解时间是否是被跳过的时间，可以通过查看从 `mktimed()` 获得的 `tm_mday`、`tm_hour` 和 `tm_min` 值是否与输入的相同来检查。仅检查 `tm_hour` 和 `tm_min` 可能看起来足够，但 `tm_mday` 也可能更改——而不更改 `tm_hour` 和 `tm_min`——如果例如 `TZ` 设置为"ABC12XYZ-12"（可能用于压力测试）或如果地理时区将其标准时间与协调世界时的偏移量更改了 24 小时。

理由

为了允许应用程序区分 `(time_t)-1` 的成功返回和 [EOVERFLOW] 错误，`mktimed()` 被要求在错误时不更改 `tm_wday`。使用这种机制而不是其他函数使用的约定（应用程序在调用前将 `errno` 设置为零，调用在错误时不更改 `errno`），因为 ISO C 标准不要求 `mktimed()` 在错误时设置 `errno`。ISO C 标准的下一修订版预计要求 `mktimed()` 在返回 `(time_t)-1` 表示错误时不更改 `tm_wday`，并且此返回约定既用于函数返回值不能表示为 `time_t` 的情况，也用于要在 `tm` 结构体的 `tm_year` 成员中返回的值不能表示为 `int` 的情况。

`DESCRIPTION` 部分说 `mktimed()` 将指定的分解时间转换为一个自纪元以来的时间值。此处使用不定冠词是必要的，因为当 `tm_isdst` 为负数且时区有夏令时转换时，分解时间与自纪元以来的时间值之间不存在一一对应的关系。

`tm_isdst` 的值如何影响 `mktimed()` 行为的描述被标记为 CX，因为 ISO C 标准中的要求不明确。ISO C 标准的下一修订版预计使用与此标准中等效的措辞来说明要求。

鼓励实现不范围修正 `tm_sec`（参见 `DESCRIPTION` 中的步骤1），以便使对 `tm_sec` 进行调整的结果始终等同于对 `mktimed()` 返回值进行相同的调整，即使原始时间和更新时间位于地理时区更改的不同侧。这为应用程序提供了一种仅使用 `mktimed()` 进行可靠固定时间段调整的方法，如应用程序用法中所述。

所描述的范围修正 `tm` 结构体成员的方法使用单独的变量保存要稍后应用到其他成员的调整值，或（对于年份调整）在后续计算中使用，因为这是避免中间成

员值无法表示为 `int` 的一种方法。实现可以使用其他方法；唯一的要求是 `tm_year` 是唯一可能发生 [EOVERFLOW] 错误的成员。

如果按所描述的方式实现，所描述的范围修正 `tm_mday` 的方法对于非常大的值将非常低效。可以通过观察到任何 400 年的周期总是具有相同的天数来提高效率，因此逐月修正方法最多只需要应用于 4800 个月。

未来方向

本标准的未来版本可能要求 `mktimed()` 不执行 DESCRIPTION 中步骤1描述的 `tm` 结构体的 `tm_sec` 成员的可选范围修正。

本标准的未来版本预计添加一个类似于 `mktimed()` 的 `timegm()` 函数，不同之处在于 `timeptr` 指向的 `tm` 结构体包含协调世界时（而不是本地时区）的分解时间，其中对 `localtime()` 的引用被替换为对 `gmtime()` 的引用，并且没有时区偏移量或夏令时调整。`gmtime()` 和 `timegm()` 的组合将是预期的方法来对 `time_t` 值执行算术运算并与 ISO C 标准保持兼容（其中 `time_t` 的内部结构未指定），因为尝试使用 `localtime()` 和 `mktimed()` 进行此类操作可能导致意外结果。

参见

`asctime()`, `clock()`, `ctime()`, `difftime()`, `futimens()`,
`gmtime()`, `localtime()`, `strftime()`, `strptime()`, `time()`,
`tzset()`

XBD 4.19 自纪元以来的秒数, `<time.h>`

更改历史

首次发布于 Issue 3。包含以与 POSIX.1-1988 标准和 ANSI C 标准对齐。

Issue 6

超出 ISO C 标准的扩展被标记。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/58，更新返回值和错误部分以将可选的 [EOVERFLOW] 错误添加为 CX 扩展。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/59，将 `tzset()` 函数添加到参见部分。

Issue 7

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0393 [104]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0228 [724]。

Issue 8

应用 Austin Group Defect 1253, 将 "Daylight Savings" 更改为 "Daylight Saving"。

应用 Austin Group Defect 1613, 更改 `mktime()` 使用的 **tm** 结构体成员的指定方式, 并阐明成功调用将成员设置为与 `localtime()` 将返回的相同值。

应用 Austin Group Defect 1614, 阐明 `tm_isdst` 的处理方式以及 `(time_t)-1` 返回的条件。

应用 Austin Group Defect 1627, 阐明 `mktime()` 如何从 **tm** 结构体的成员计算自纪元以来的秒数时间。

1.95. mlock, munlock — 锁定或解锁进程地址空间范围 (REALTIME)

概要

```
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

描述

`mlock()` 函数应使包含进程地址空间任何部分的整个页面（从地址 `addr` 开始，持续 `len` 字节）常驻内存，直到被解锁、进程退出或 `exec` 另一个进程映像为止。实现可能要求 `addr` 是 {PAGESIZE} 的倍数。

`munlock()` 函数应解锁包含进程地址空间任何部分的整个页面（从地址 `addr` 开始，持续 `len` 字节），无论进程对指定范围内的任何页面调用了多少次 `mlock()`。实现可能要求 `addr` 是 {PAGESIZE} 的倍数。

如果指定给 `munlock()` 调用范围内的任何页面也被映射到其他进程的地址空间中，那么另一个进程在这些页面上建立的任何锁定都不会受到此进程调用 `munlock()` 的影响。如果指定给 `munlock()` 调用的范围内的任何页面也被映射到调用进程地址空间中指定范围之外的其他部分，那么通过其他映射在这些页面上建立的任何锁定也不会受到此次调用的影响。

成功从 `mlock()` 返回时，指定范围内的页面应被锁定并常驻内存。成功从 `munlock()` 返回时，指定范围内的页面应相对于进程的地址空间被解锁。已解锁页面的内存常驻性未指定。

使用 `mlock()` 锁定进程内存需要适当的权限。

返回值

成功完成时，`mlock()` 和 `munlock()` 函数应返回零值。否则，进程地址空间中的任何锁定都不会被更改，函数应返回 -1 值并设置 `errno` 来指示错误。

错误

`mlock()` 和 `munlock()` 函数在以下情况下应失败：

- **[ENOMEM]**

`addr` 和 `len` 参数指定的地址范围的某些或全部不对应于进程地址空间中的有效映射页面。

`mlock()` 函数在以下情况下应失败：

- **[EAGAIN]**

进行调用时，操作所标识的某些或全部内存无法被锁定。

`mlock()` 和 `munlock()` 函数在以下情况下可能失败：

- **[EINVAL]**

`addr` 参数不是 {PAGESIZE} 的倍数。

`mlock()` 函数在以下情况下可能失败：

- **[ENOMEM]**

锁定指定范围映射的页面将超过进程可锁定内存量的实现定义限制。

- **[EPERM]**

调用进程没有执行所请求操作的适当权限。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

参见

- `exec`
- `_exit()`
- `fork()`
- `mlockall()`
- `munmap()`
- `<sys/mman.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 实时扩展保持一致而包含。

Issue 6

`mlock()` 和 `munlock()` 函数被标记为范围内存锁定选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持范围内存锁定选项，则无需提供存根。

1.96. mlockall

概要

```
#include <sys/mman.h>

int mlockall(int flags);
int munlockall(void);
```

描述

`mlockall()` 函数应使进程地址空间映射的所有页面在内存中保持驻留，直到被解锁、进程退出或 `exec` 执行另一个进程映像为止。`flags` 参数决定锁定的页面是当前由进程地址空间映射的页面、将来要映射的页面，还是两者兼有。`flags` 参数由以下一个或多个符号常量的按位包含 OR 构成，这些常量定义在 `<sys/mman.h>` 中：

- **MCL_CURRENT**

锁定当前映射到进程地址空间的所有页面。

- **MCL_FUTURE**

锁定将来映射到进程地址空间的所有页面，当这些映射建立时锁定。

如果指定了 `MCL_FUTURE`，并且对未来映射的自动锁定最终导致锁定的内存量超过可用物理内存的量或任何其他实现定义的限制，则行为是实现定义的。实现通知应用程序这些情况的方式也是实现定义的。

`munlockall()` 函数应解锁进程地址空间的所有当前映射页面。在调用 `munlockall()` 之后映射到进程地址空间的任何页面都不应被锁定，除非有中间的 `mlockall()` 调用指定了 `MCL_FUTURE`，或后续的 `mlockall()` 调用指定了 `MCL_CURRENT`。如果映射到进程地址空间的页面也映射到其他进程的地址空间并被这些进程锁定，则其他进程建立的锁定不受此进程调用 `munlockall()` 的影响。

在成功返回指定 `MCL_CURRENT` 的 `mlockall()` 函数后，进程地址空间的所有当前映射页面应在内存中驻留并被锁定。从 `munlockall()` 函数返回后，进程地址空间的所有当前映射页面应相对于进程地址空间被解锁。未锁定页面的内存驻留情况是未指定的。

使用 `mlockall()` 锁定进程内存需要适当的权限。

返回值

成功完成后, `mlockall()` 函数应返回零值。否则, 不应锁定额外的内存, 函数应返回值 -1 并设置 `errno` 以指示错误。`mlockall()` 失败对地址空间中先前存在的锁定的影响是未指定的。

如果实现支持, `munlockall()` 函数应始终返回零值。否则, 函数应返回值 -1 并设置 `errno` 以指示错误。

错误

`mlockall()` 函数在以下情况下应失败:

- **[EAGAIN]**

调用时操作标识的部分或全部内存无法被锁定。

- **[EINVAL]**

`flags` 参数为零, 或包含未实现的标志。

`mlockall()` 函数在以下情况下可能失败:

- **[ENOMEM]**

锁定当前映射到进程地址空间的所有页面将超过进程可锁定内存量的实现定义限制。

- **[EPERM]**

调用进程没有执行所请求操作的适当权限。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参见

`exec` , `exit()` , `fork()` , `mlock()` , `munmap()`

XBD `<sys/mman.h>`

变更历史

首次发布于 Issue 5。为了与 POSIX 实时扩展对齐而包含。

Issue 6

`mlockall()` 和 `munlockall()` 函数被标记为进程内存锁定选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持进程内存锁定选项，则不需要提供存根。

1.97. mmap

概要

```
#include <sys/mman.h>

void *mmap(void *addr, size_t len, int prot, int flags,
           int fildes, off_t off);
```

描述

`mmap()` 函数应在进程的地址空间和内存对象之间建立映射关系。

`mmap()` 函数应支持以下内存对象：

- 常规文件
- 匿名内存对象
- [SHM] 共享内存对象
- [TYM] 类型化内存对象

对于任何其他类型的文件的支持是未指定的。

调用的格式如下：

```
pa = mmap(addr, len, prot, flags, fildes, off);
```

`mmap()` 函数应在进程地址空间的地址 `pa` 处为 `len` 字节到由文件描述符 `fildes` 表示的内存对象的偏移量 `off` 处的 `len` 字节之间建立映射，或者到 `len` 字节的匿名内存对象。`pa` 的值是参数 `addr` 和 `flags` 值的实现定义函数，进一步描述如下。成功的 `mmap()` 调用应返回 `pa` 作为其结果。从 `pa` 开始并继续 `len` 字节的地址范围对于进程的可能（不一定当前）地址空间应是合法的。从 `off` 开始并继续 `len` 字节的字节范围对于由 `fildes` 表示的内存对象的可能（不一定当前）偏移量应是合法的。

[TYM] 如果 `fildes` 表示使用 `POSIX_TYPED_MEM_ALLOCATE` 标志或 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` 标志打开的类型化内存对象，要映射的内存对象应是实现按如下规定分配的类型化内存对象的那部分。在这种情况下，如果 `off` 非零，`mmap()` 的行为是未定义的。如果 `fildes` 引用从调用进程不可访问的有效类型化内存对象，`mmap()` 应失败。

由 `mmap()` 建立的映射应替换包含从 `pa` 开始并继续 `len` 字节的进程地址空间任何部分的那些整个页面的任何先前映射。

如果在调用 `mmap()` 后映射文件的大小由于对映射文件的某些其他操作而改变，对映射区域中对应于文件添加或删除部分的引用的影响是未指定的。

如果 `len` 为零，`mmap()` 应失败且不建立映射。

参数 `prot` 决定是否允许对要映射的数据进行读、写、执行或某种组合访问。

`prot` 应是 `PROT_NONE` 或下表中在 `<sys/mman.h>` 头文件中定义的其他一个或多个标志的按位包含或。

符号常量	描述
<code>PROT_READ</code>	数据可读取。
<code>PROT_WRITE</code>	数据可写入。
<code>PROT_EXEC</code>	数据可执行。
<code>PROT_NONE</code>	数据不可访问。

如果实现无法支持 `prot` 指定的访问类型组合，`mmap()` 调用应失败。

实现可以允许除 `prot` 指定的访问之外的访问；但是，实现不应允许在未设置 `PROT_WRITE` 时写入成功，也不应在仅设置 `PROT_NONE` 时允许任何访问。实现应至少支持以下 `prot` 值：`PROT_NONE`、`PROT_READ`、`PROT_WRITE` 以及 `PROT_READ` 和 `PROT_WRITE` 的按位包含或。文件描述符 `fildes` 应已以读权限打开，无论指定的保护选项如何。如果指定了 `PROT_WRITE`，应用程序应确保已以写权限打开文件描述符 `fildes`，除非在 `flags` 参数中指定了 `MAP_PRIVATE`，如下所述。

参数 `flags` 提供关于处理映射数据的其他信息。`flags` 的值是这些选项的按位包含或，在 `<sys/mman.h>` 中定义：

符号常量	描述
<code>MAP_ANON</code>	<code>MAP_ANONYMOUS</code> 的同义词。
<code>MAP_ANONYMOUS</code>	映射匿名内存。
<code>MAP_SHARED</code>	更改是共享的。

符号常量	描述
MAP_PRIVATE	更改是私有的。
MAP_FIXED	精确解释 addr。

是否支持 MAP_FIXED 是实现定义的。[XSI] 在符合 XSI 的系统上应支持 MAP_FIXED。

MAP_SHARED 和 MAP_PRIVATE 描述对内存对象的写引用的处理。如果指定了 MAP_SHARED，写引用应更改基础对象。如果指定了 MAP_PRIVATE，调用进程对映射数据的修改仅对调用进程可见，且不应更改基础对象。在建立 MAP_PRIVATE 映射后，对基础对象的修改是否通过 MAP_PRIVATE 映射可见是未指定的。可以指定 MAP_SHARED 或 MAP_PRIVATE，但不能同时指定两者。

映射类型在 `fork()` 调用中保持不变。

当任何进程中包含同步对象的最后一个区域取消映射时，放置在使用 MAP_SHARED 映射的共享内存中的互斥锁、信号量、屏障和条件变量等同步对象的状态变为未定义。

[TYM] 当 `fd` 表示使用 `POSIX_TYPED_MEM_ALLOCATE` 标志或 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` 标志打开的类型化内存对象时，`mmap()` 应在有足够的可用资源的情况下，映射从相应类型化内存对象分配的 `len` 字节，这些字节先前未分配给可能访问该类型化内存对象的任何处理器中的任何进程。如果没有足够的可用资源，函数应失败。如果 `fd` 表示使用 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` 标志打开的类型化内存对象，这些分配的字节在类型化内存对象内应是连续的。如果 `fd` 表示使用 `POSIX_TYPED_MEM_ALLOCATE` 标志打开的类型化内存对象，这些分配的字节可能由类型化内存对象内的非连续片段组成。如果 `fd` 表示既未使用 `POSIX_TYPED_MEM_ALLOCATE_CONTIG` 标志也未使用 `POSIX_TYPED_MEM_ALLOCATE` 标志打开的类型化内存对象，则映射类型化内存对象内从偏移量 `off` 开始的 `len` 字节，与映射文件或共享内存对象时完全相同。在这种情况下，如果两个进程使用相同的 `off` 和 `len` 值以及引用相同内存池（来自同一端口或不同端口）的文件描述符映射类型化内存的区域，两个进程应映射相同的存储区域。

在 `flags` 参数中设置 MAP_FIXED 时，通知实现 `pa` 的值应完全是 `addr`。如果设置了 MAP_FIXED，`mmap()` 可能返回 MAP_FAILED 并将 `errno` 设置为 [EINVAL]。如果 MAP_FIXED 请求成功，则在建立新映射之前，应删除包含地址范围 `[pa, pa + len]` 任何部分的那些整个页面的任何先前映射 [ML|MLR] 或内存锁定，就像通过适当的 `munmap()` 调用一样。

当未设置 MAP_FIXED 时，实现以实现定义的方式使用 `addr` 来得出 `pa`。这样选择的 `pa` 应是实现认为适合将 `len` 字节映射到文件的地址空间区域。所有实现将 `addr` 值 0 解释为授予实现在选择 `pa` 时的完全自由，但要受下面描述的约束。`addr` 的非零值被看作是放置映射的进程地址的建议。当实现为 `pa` 选择值时，它永远不会将映射放置在地址 0 处，也不会替换任何现有映射。

如果指定了 MAP_FIXED 且 `addr` 非零，它应具有与 `off` 参数相同的余数，以页大小为模，该页大小通过向 `sysconf()` 传递 `_SC_PAGESIZE` 或 `_SC_PAGE_SIZE` 返回。实现可能要求 `off` 是页大小的倍数。如果指定了 MAP_FIXED，实现可能要求 `addr` 是页大小的倍数。系统对整个页面执行映射操作。因此，虽然参数 `len` 不需要满足大小或对齐约束，但系统应在任何映射操作中包含从 `pa` 开始并继续 `len` 字节的地址范围指定的任何部分页面。

如果指定了 MAP_ANONYMOUS (或其同义词 MAP_ANON)，`fildes` 为 -1，`off` 为 0，则 `mmap()` 应忽略 `fildes`，而是建立到大小为 `len` 的新匿名内存对象的映射。将 MAP_ANONYMOUS (或 MAP_ANON) 与 `fildes` 或 `off` 的其他值一起指定的效果是未指定的。匿名内存对象应初始化为全零。

系统应始终在对象末尾用零填充任何部分页面。此外，系统永远不会写出对象最后一页超出其末尾的任何修改部分。在从 `pa` 开始并继续 `len` 字节的地址范围内对对象末尾之后的整个页面的引用应导致传送 SIGBUS 信号。

当引用会导致映射对象中的错误 (如空间不足条件) 时，实现可能生成 SIGBUS 信号。

`mmap()` 函数应向与文件描述符 `fildes` 关联的文件添加额外引用，该引用不会通过后续对该文件描述符的 `close()` 调用来移除。当文件不再有映射时，应移除此引用。

映射文件的最后数据访问时间戳可能在 `mmap()` 调用和相应的 `munmap()` 调用之间的任何时间被标记为更新。对映射区域的初始读或写引用应导致文件的最后数据访问时间戳被标记为更新 (如果尚未被标记为更新)。

使用 MAP_SHARED 和 PROT_WRITE 映射的文件的最后数据修改和最后文件状态更改时间戳应在对映射区域的写引用和任何进程对该部分文件的下一次使用 `MS_ASYNC` 或 `MS_SYNC` 调用 `msync()` 之间的某个时间点被标记为更新。如果没有这样的调用，并且如果基础文件由于写引用而被修改，那么这些时间戳应在写引用后的某个时间被标记为更新。

可能存在实现定义的限制，限制可映射的内存区域数量 (每个进程或每个系统)。

[XSI] 如果施加了这样的限制，进程可映射的内存区域数量是否因使用 `shmat()` 而减少是实现定义的。

如果 `mmap()` 因 [EBADF]、[EINVAL] 或 [ENOTSUP] 以外的原因失败，从 `addr` 开始并继续 `len` 字节的地址范围内的一些映射可能已被取消映射。

返回值

成功完成后，`mmap()` 函数应返回映射放置的地址（`pa`）；否则，应返回 `MAP_FAILED` 值并设置 `errno` 以指示错误。符号 `MAP_FAILED` 在 `<sys/mman.h>` 头文件中定义。`mmap()` 的任何成功返回都不应返回 `MAP_FAILED` 值。

错误

在以下情况下，`mmap()` 函数应失败：

[EAGAIN]

[ML] 如果 `mlockall()` 要求，由于资源缺乏，映射无法锁定在内存中。

[EINVAL]

`len` 的值为零。

[EINVAL]

`flags` 的值无效（既未设置 `MAP_PRIVATE` 也未设置 `MAP_SHARED`）。

[EMFILE]

映射区域数量将超过实现定义的限制（每个进程或每个系统）。

[ENOMEM]

指定了 `MAP_FIXED`，并且范围 [`addr` , `addr` + `len`] 超过了进程地址空间允许的范围；或者，如果未指定 `MAP_FIXED`，并且地址空间中没有足够的空间来建立映射。

[ENOMEM]

[ML] 如果 `mlockall()` 要求，由于需要比系统能够提供的更多空间，映射无法锁定在内存中。

[ENOMEM]

[TYM] 由 `fd` 指定的类型化内存对象中剩余的未分配内存资源不足以分配 `len` 字节。

[ENOTSUP]

在 `flags` 参数中指定了 `MAP_FIXED` 或 `MAP_PRIVATE`，而实现不支持此功能。

实现不支持 `prot` 参数中请求的访问组合。

[ENXIO]

在 `flags` 中指定了 `MAP_FIXED`，并且 `addr`、`len` 和 `off` 的组合对于指定对象无效。

[ENXIO]

[TYM] `fildes` 参数引用从调用进程不可访问的类型化内存对象。

此外，如果 `flags` 中未设置 `MAP_ANONYMOUS` (或 `MAP_ANON`)，在以下情况下 `mmap()` 函数应失败：

[EACCES]

`fildes` 参数未以读权限打开，无论指定何种保护，或者 `fildes` 未以写权限打开，并且为 `MAP_SHARED` 类型映射指定了 `PROT_WRITE`。

[EBADF]

`fildes` 参数不是有效的打开文件描述符。

[ENODEV]

`fildes` 参数引用 `mmap()` 不支持的文件类型。

[EOVERFLOW]

文件是常规文件，并且 `off` 加 `len` 的值超过与 `fildes` 关联的打开文件描述中建立的偏移量最大值。

[ENXIO]

对于 `fildes` 指定的对象，范围 $[off, off + len]$ 中的地址无效。

在以下情况下，`mmap()` 函数可能失败：

[EINVAL]

`addr` 参数 (如果指定了 `MAP_FIXED`) 或 `off` 不是 `sysconf()` 返回的页大小的倍数，或者被实现认为是无效的。

示例

无。

应用程序用法

使用 `mmap()` 可能会减少可用于其他内存分配函数的内存量。

使用 `MAP_FIXED` 可能导致在进一步使用 `malloc()` 和 `shmat()` 时出现未定义行为。不鼓励使用 `MAP_FIXED`，因为它可能阻止实现最有效地利用资源。大

多数实现要求 `off` 和 `addr` 是 `sysconf()` 返回的页大小的倍数。

当将 `mmap()` 与任何其他文件访问方法（如 `read()` 和 `write()`、标准输入/输出和 `shmat()`）结合使用时，应用程序必须确保正确的同步。

`mmap()` 函数允许通过地址空间操作而不是 `read()` / `write()` 来访问资源。一旦文件被映射，进程访问它所需要做的就是使用文件映射到的地址处的数据。因此，使用伪代码来说明现有程序可能更改为使用 `mmap()` 的方式，如下所示：

```
fildes = open(...)  
lseek(fildes, some_offset)  
read(fildes, buf, len)  
/* 使用 buf 中的数据。 */
```

变为：

```
fildes = open(...)  
address = mmap(0, len, PROT_READ, MAP_PRIVATE, fildes, some_offset)  
/* 使用 address 处的数据。 */
```

原理

在考虑了几个其他替代方案后，决定采用在 SVR4 中找到的 `mmap()` 定义，用于将内存对象映射到进程地址空间。SVR4 定义是最小的，因为它只描述了已构建的内容，以及对于通用和可移植映射设施看起来必要的内容。

请注意，虽然 `mmap()` 最初是为映射文件设计的，但它实际上是一个通用映射设施。它可用于将任何适当的对象（如内存、文件、设备等）映射到进程的地址空间。

建立映射时，由于硬件要求，实现可能需要在进程地址空间中映射比请求更多的内容。但是，应用程序不能依赖这种行为。不使用分页架构的实现可能只是分配一个公共内存区域并返回其地址；这样的实现可能不会分配超出必要的内容。对请求区域末尾之外的引用是未指定的。

如果应用程序请求的映射与进程中的现有映射重叠，可能希望实现检测到这一点并通知应用程序。但是，如果程序指定了固定地址映射（如果支持该功能，需要一些实现知识来确定合适的地址），那么程序被假定为成功地管理自己的地址空间，当请求映射到现有数据结构上时应该被信任。此外，还希望尽可能少地进行系统调用，在相同地址范围的 `mmap()` 之前要求 `munmap()` 可能被认为是繁重的。POSIX.1-2024 卷指定新映射替换任何现有映射（意味着对地址范

围的自动 `munmap()`），遵循这方面的现有实践。标准开发者还考虑了是否应该有一种方式让新映射覆盖现有映射，但没有发现这方面的现有实践。

并不期望所有硬件实现都能在所有地址上支持所有权限组合。要求实现禁止对没有写权限的映射进行写访问，并禁止对没有任何访问权限的映射进行任何访问。除了这些限制外，实现可以允许应用程序请求的访问类型之外的访问类型。例如，如果应用程序仅请求 `PROT_WRITE`，实现也可以允许读访问。如果实现无法支持允许应用程序请求的所有访问，`mmap()` 调用失败。例如，一些实现不能同时支持写访问和执行访问的请求。所有实现必须支持无访问、读访问、写访问以及读和写访问的请求。严格符合的代码只能依赖所需的检查。这些限制允许在各种硬件上具有可移植性。

`MAP_FIXED` 地址处理对于非页对齐值和某些依赖架构的地址范围可能会失败。符合的实现不能指望能够在不利用不可移植、实现定义的知识的情况下为 `MAP_FIXED` 选择地址值。尽管如此，`MAP_FIXED` 作为符合现有实践的标准接口提供，用于在可用时利用此类知识。

类似地，为了允许不支持虚拟地址的实现，不要求支持通过 `MAP_FIXED` 直接指定任何映射地址，因此符合的应用程序可能不能依赖它。

当内存保护硬件可用时，`MAP_PRIVATE` 函数可以高效实现。当这样的硬件不可用时，实现可以通过简单地将相关数据的实际副本制作到进程私有内存中来实现此类“映射”，尽管这往往与 `read()` 表现相似。

该函数已被定义为允许许多不同的使用共享内存的模型。但是，并非所有用途在所有机器架构上都同样可移植。特别是，`mmap()` 函数允许系统和应用程序指定将内存对象的特定区域映射到的地址。使用函数的最可移植方式始终是让系统选择地址，指定 `NULL` 作为 `addr` 参数的值，并且不指定 `MAP_FIXED`。

如果打算将内存对象的特定区域映射到一组进程中的相同地址（在甚至可能的机器上），那么可以使用 `MAP_FIXED` 传入所需的映射地址。如果第一次这样的映射是在未指定 `MAP_FIXED` 的情况下进行的，系统仍然可以用来选择所需地址，然后生成的映射地址可以传递给后续进程，以便它们通过 `MAP_FIXED` 传入。通常不能保证特定地址范围的可用性。

`mmap()` 函数可用于映射大于对象当前大小的内存区域。在映射内但超出基础对象当前末尾的内存访问可能导致向进程发送 `SIGBUS` 信号。原因是对象的大小可能被其他进程操纵，并且可以随时更改。实现应告诉应用程序内存引用在对象之外，在可以检测到的情况下；否则，写入的数据可能会丢失，读取的数据可能不会反映对象中的实际数据。

请注意，对对象末尾之外的引用不会扩展对象，因为新的末尾无法被大多数虚拟内存硬件精确确定。相反，大小可以直接通过 `ftruncate()` 操纵。

进程内存锁定确实适用于共享内存区域，`mlockall()` 的 `MCL_FUTURE` 参数可用于确保新的共享内存区域被自动锁定。

`mmap()` 的现有实现在不成功时返回值 -1。由于 ISO C 标准无法保证将此值强制转换为 `void *` 类型与成功值不同，POSIX.1-2024 卷定义了符号 `MAP_FAILED`，符合的实现不将其作为成功调用的结果返回。

一些历史实现仅支持 `MAP_ANON`，一些仅支持 `MAP_ANONYMOUS`，一些支持两种拼写。本标准包含两种拼写，部分是为了应用程序兼容性，部分因为在考虑此功能标准化时，没有哪种拼写明显比另一种更流行。

未来方向

无。

另请参阅

`exec`、`fcntl()`、`fork()`、`lockf()`、`msync()`、`munmap()`、
`mprotect()`、`posix_TYPED_mem_open()`、`shmat()`、`sysconf()`

XBD `<sys/mman.h>`

更改历史

首次在版本 4 的第 2 版中发布。

版本 5

从 X/OPEN UNIX 扩展移动到 BASE。

与 POSIX Realtime Extension 中的 `mmap()` 对齐如下：

- 描述被广泛重写。
- 添加了 `[EAGAIN]` 和 `[ENOTSUP]` 强制错误条件。
- 添加了 `[ENOMEM]` 和 `[ENXIO]` 的新案例作为强制错误条件。
- 失败时返回的值是常量 `MAP_FAILED` 的值；这之前被定义为 -1。

添加了 Large File Summit 扩展。

版本 6

`mmap()` 函数被标记为内存映射文件选项的一部分。

应用了 The Open Group Corrigendum U028/6，将 `(void *) -1` 更改为 `MAP_FAILED`。

以下对 POSIX 实现的新要求源自与 Single UNIX Specification 的对齐：

- 更新描述以描述 `MAP_FIXED` 的使用。
- 更新描述以描述向传递给 `mmap()` 的文件描述符关联的文件添加额外引用。
- 更新描述以说明可能存在对可映射内存区域数量的实现定义限制。
- 更新描述以描述 `off` 参数的对齐和大小约束。
- 添加了 `[EINVAL]` 和 `[EMFILE]` 错误条件。
- 添加了 `[EOVERFLOW]` 错误条件。此更改是为了支持大文件。

为了与 ISO POSIX-1:1996 标准对齐，进行了以下更改：

- 更新描述以描述 `MAP_PRIVATE` 和 `MAP_FIXED` 不需要支持的情况。

为了与 IEEE Std 1003.1j-2000 对齐，进行了以下更改：

- 在描述中添加了类型化内存对象的语义。
- 在错误部分添加了新的 `[ENOMEM]` 和 `[ENXIO]` 错误。
- 将 `posix_typed_mem_open()` 函数添加到"另请参阅"部分。

更新规范文本以避免对应用程序要求使用"必须"一词。

应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/34，将概要中的边距代码从 `MF|SHM` 更改为 `MC3` (`MF|SHM|TYM` 的表示法)。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/60，更新描述和错误部分以在 `len` 为零时添加 `[EINVAL]` 错误。

版本 7

应用了 Austin Group Interpretations 1003.1-2001 #078 和 #079，阐明了页对齐要求，并添加了关于当共享区域取消映射时同步对象状态变为未定义的说明。

与内存保护和内存映射文件选项相关的功能被移动到基础。

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008，Technical Corrigendum 2，XSH/TC2-2008/0229 [852]。

版本 8

应用了 Austin Group Defect 850，添加了匿名内存对象。

1.98. munlock

概要 (SYNOPSIS)

```
[MLR]
#include <sys/mman.h>

int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
```

描述 (DESCRIPTION)

`mlock()` 函数应导致包含从地址 `addr` 开始、持续 `len` 字节的进程地址空间任意部分的所有整页，在解锁之前或进程退出或 `exec` 另一个进程映像之前常驻内存。实现可能要求 `addr` 是 {PAGESIZE} 的倍数。

`munlock()` 函数应解锁包含从地址 `addr` 开始、持续 `len` 字节的进程地址空间任意部分的所有整页，无论进程已对指定范围内的任何页面调用了多少次 `mlock()`。实现可能要求 `addr` 是 {PAGESIZE} 的倍数。

如果在一次 `munlock()` 调用中指定的范围内有任何页面也映射到了其他进程的地址空间中，那么由其他进程在这些页面上建立的任何锁定都不会受到此进程调用 `munlock()` 的影响。如果在一次 `munlock()` 调用中指定的范围内有任何页面也映射到了调用进程地址空间中指定范围之外的其他部分，那么通过其他映射在这些页面上建立的任何锁定也不会受到此调用的影响。

从 `mlock()` 成功返回时，指定范围内的页面应被锁定并常驻内存。从 `munlock()` 成功返回时，指定范围内的页面应相对于进程的地址空间被解锁。解锁页面的内存驻留性是未指定的。

使用 `mlock()` 锁定进程内存需要适当的权限。

返回值 (RETURN VALUE)

成功完成时，`mlock()` 和 `munlock()` 函数应返回值为零。否则，进程地址空间中的任何锁定都不会发生更改，函数应返回值为 -1 并设置 `errno` 来指示错误。

错误 (ERRORS)

`mlock()` 和 `munlock()` 函数在以下情况下应失败：

- **[ENOMEM]**

由 `addr` 和 `len` 参数指定的地址范围的某些或全部不对应于进程地址空间中的有效映射页面。

`mlock()` 函数在以下情况下应失败：

- **[EAGAIN]**

在进行调用时，操作所标识的某些或全部内存无法被锁定。

`mlock()` 和 `munlock()` 函数在以下情况下可能失败：

- **[EINVAL]**

`addr` 参数不是 `{PAGESIZE}` 的倍数。

`mlock()` 函数在以下情况下可能失败：

- **[ENOMEM]**

锁定由指定范围映射的页面将超过实现定义的进程可锁定内存量的限制。

- **[EPERM]**

调用进程没有执行请求操作的适当权限。

以下各节为参考性内容。

示例 (EXAMPLES)

无。

应用程序使用 (APPLICATION USAGE)

无。

原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

`exec()` , `_exit()` , `fork()` , `mlockall()` , `munmap()`

XBD `<sys/mman.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 5。为与 POSIX 实时扩展保持一致而包含在内。

Issue 6

`mlock()` 和 `munlock()` 函数被标记为范围内存锁定选项的一部分。

如果实现不支持范围内存锁定选项，则不需要提供存根，因此 [ENOSYS] 错误条件已被移除。

参考性文本结束。

1.99. munmap — 取消内存页映射

SYNOPSIS 概要

```
#include <sys/mman.h>

int munmap(void *addr, size_t len);
```

DESCRIPTION 描述

`munmap()` 函数应当移除那些包含从 `addr` 开始、持续 `len` 字节的进程地址空间任何部分的整个页面的任何映射。对这些页面的进一步引用应当导致向进程生成 SIGSEGV 信号。如果在指定的地址范围内没有映射，那么 `munmap()` 将不起任何作用。

实现可能要求 `addr` 是 `sysconf()` 返回的页面大小的倍数。

如果要移除的映射是私有的，那么在此地址范围内所做的任何修改都将被丢弃。

[ML|MLR] 与此地址范围关联的任何内存锁（参见 `mlock()` 和 `mlockall()`）都应当被移除，就像通过适当的 `munlock()` 调用一样。

[TYM] 如果从类型化内存对象中移除映射导致内存池的相应地址范围除了通过可分配映射（即使用 POSIX_TYPED_MEM_MAP_ALLOCATABLE 标志打开的类型化内存对象的映射）之外，对系统中的任何进程都不可访问，那么该内存池的范围应当被解除分配，并可能变得可用于满足未来的类型化内存分配请求。

从使用 POSIX_TYPED_MEM_MAP_ALLOCATABLE 标志打开的类型化内存对象中移除的映射不应当以任何方式影响该类型化内存用于分配的可用性。

如果映射不是通过 `mmap()` 调用建立的，则此函数的行为是未指定的。

RETURN VALUE 返回值

成功完成后，`munmap()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

ERRORS 错误

`munmap()` 函数应在以下情况下失败：

- **[EINVAL]** 范围 [`addr` , `addr` + `len`) 内的地址超出了进程地址空间的有效范围。
- **[EINVAL]** `len` 参数为 0。

`munmap()` 函数可能在以下情况下失败：

- **[EINVAL]** `addr` 参数不是 `sysconf()` 返回的页面大小的倍数。

EXAMPLES 示例

无。

APPLICATION USAGE 应用程序用法

无。

RATIONALE 基本原理

`munmap()` 函数对应于 SVR4，就像 `mmap()` 函数一样。

应用程序可能已经对包含共享内存的区域应用了进程内存锁定。如果发生了这种情况，`munmap()` 调用会忽略这些锁，并在必要时导致这些锁被移除。

大多数实现要求 `addr` 是 `sysconf()` 返回的页面大小的倍数。

FUTURE DIRECTIONS 未来方向

无。

SEE ALSO 另请参见

- `mlock()`
- `mlockall()`
- `mmap()`

- `posix_typed_mem_open()`
- `sysconf()`
- XBD `<sys/mman.h>`

CHANGE HISTORY 变更历史

首次发布于 Issue 4, Version 2。

Issue 5

从 X/OPEN UNIX 扩展移至 BASE。

与 POSIX Realtime Extension 中的 `munmap()` 对齐如下：

- DESCRIPTION 进行了大量重写。
- 不再允许生成 SIGBUS 错误。

Issue 6

`munmap()` 函数被标记为内存映射文件和共享内存对象选项的一部分。

POSIX 实现的以下新要求来源于与 Single UNIX Specification 的对齐：

- DESCRIPTION 更新为声明实现要求 `addr` 是页面大小的倍数。
- 添加了 [EINVAL] 错误条件。

为与 IEEE Std 1003.1j-2000 对齐进行了以下更改：

- 类型化内存对象的语义被添加到 DESCRIPTION 中。
- `posix_typed_mem_open()` 函数被添加到 SEE ALSO 部分。

应用 IEEE Std 1003.1-2001/Cor 1-2002, 项目 XSH/TC1/D6/36, 将 SYNOPSIS 中的边距代码从 MF|SHM 更改为 MC3 (MF|SHM|TYM 的表示法)。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #078, 阐明了页面对齐要求。

`munmap()` 函数从内存映射文件选项移至 Base。

1.100. nanosleep — 高精度睡眠

SYNOPSIS

```
#include <time.h>

int nanosleep(const struct timespec *rqtp, struct timespec *rmt
```

DESCRIPTION

`nanosleep()` 函数应导致当前线程暂停执行，直到由 `rqtp` 参数指定的时间间隔已经过去，或者向调用线程发送了一个信号，且该信号的动作是调用信号捕获函数或终止进程。暂停时间可能比请求的时间更长，因为参数值会向上舍入为睡眠分辨率的整数倍，或者因为系统对其他活动的调度。但是，除了被信号中断的情况外，暂停时间不应小于由 `rqtp` 指定的时间，这是由系统时钟 `CLOCK_REALTIME` 测量的。

使用 `nanosleep()` 函数对任何信号的动作或阻塞没有影响。

RETURN VALUE

如果 `nanosleep()` 函数因为请求的时间已过去而返回，其返回值应为零。

如果 `nanosleep()` 函数因为被信号中断而返回，它应返回值 -1 并设置 `errno` 来指示中断。如果 `rmt` 参数非 NULL，它所引用的 `timespec` 结构会被更新以包含时间间隔中剩余的时间量（请求时间减去实际睡眠的时间）。`rqtp` 和 `rmt` 参数可以指向同一个对象。如果 `rmt` 参数为 NULL，则不返回剩余时间。

如果 `nanosleep()` 失败，它应返回值 -1 并设置 `errno` 来指示错误。

ERRORS

`nanosleep()` 函数在以下情况应失败：

- [EINTR]

`nanosleep()` 函数被信号中断。

- [EINVAL]

`rqtp` 参数指定的纳秒值小于零或大于等于 1000 百万。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

为了轮询非中断函数的状态，常常需要暂停线程的执行。大量实际需求可以通过对 `sleep()` 的简单扩展来满足，该扩展提供更精细的分辨率。

在 POSIX.1-1990 标准和 SVR4 中，可以实现这样的例程，但唤醒的频率受到 `alarm()` 和 `sleep()` 函数分辨率的限制。在 4.3 BSD 中，可以使用无静态存储且不保留系统设施的方式编写这样的例程。虽然可以使用剩余的 `timer_*` 函数编写与 `sleep()` 功能相似的函数，但这样的函数需要使用信号并保留一些信号编号。POSIX.1-2024 要求 `nanosleep()` 不干扰信号功能。

`nanosleep()` 函数在成功时应返回值 0，在失败或被中断时应返回 -1。后一种情况与 `sleep()` 不同。这样做是因为剩余时间通过参数结构指针 `rmtp` 返回，而不是作为返回值返回。

FUTURE DIRECTIONS

无。

SEE ALSO

- `clock_nanosleep()`
- `sleep()`
- `<time.h>`

CHANGE HISTORY

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

- `nanosleep()` 函数被标记为定时器选项的一部分。
- 移除了 [ENOSYS] 错误条件，因为如果实现不支持定时器选项，则无需提供存根。
- 应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/37，更新 SEE ALSO 部分以包含 `clock_nanosleep()` 函数。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/63，更正了 RATIONALE 部分的文本。

Issue 7

- 应用了 SD5-XBD-ERN-33。
 - `nanosleep()` 函数从定时器选项移至基础。
 - 应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0231 [909]。
-

1.101. open, openat — 打开文件

概要

```
#include <sys/stat.h>
#include <fcntl.h>

int open(const char *path, int oflag, ...);
int openat(int fd, const char *path, int oflag, ...);
```

描述

`open()` 函数用于建立文件与文件描述符之间的连接。它创建一个引用文件的打开文件描述，以及一个引用该打开文件描述的文件描述符。文件描述符被其他 I/O 函数用来引用该文件。`path` 参数指向为文件命名的路径名。

`open()` 函数应返回指定文件的文件描述符，按照 [2.6 File Descriptor Allocation](#) 中描述的方式分配。打开文件描述是新的，因此文件描述符不应与系统中任何其他进程共享它。与新文件描述符关联的 `FD_CLOEXEC` 文件描述符标志应被清除，除非在 `oflag` 中设置了 `O_CLOEXEC` 标志。与新文件描述符关联的 `FD_CLOFORK` 文件描述符标志应被清除，除非在 `oflag` 中设置了 `O_CLOFORK` 标志。

用于标记文件内当前位置的文件偏移量应设置为文件开头。

打开文件描述的文件状态标志和文件访问模式应根据 `oflag` 的值进行设置。

`oflag` 的值通过以下列表中标志的按位包含或来构造，这些标志在 `<fcntl.h>` 中定义。应用程序应在 `oflag` 的值中精确指定以下前五个值（文件访问模式）中的一个：

文件访问模式

- `O_EXEC`: 仅用于执行（非目录文件）。如果 `path` 命名一个目录且 `O_EXEC` 与 `O_SEARCH` 不是同一个值，`open()` 应失败。
- `O_RDONLY`: 仅用于读取。
- `O_RDWR`: 用于读取和写入。如果 `path` 命名一个 FIFO，且实现不支持同时为读取和写入打开 FIFO，则 `open()` 应失败。

- **O_SEARCH**: 仅用于搜索目录。如果 `path` 命名一个非目录文件且 `O_SEARCH` 与 `O_EXEC` 不是同一个值, `open()` 应失败。
- **O_WRONLY**: 仅用于写入。

附加标志

可以使用以下标志的任何组合:

- **O_APPEND**: 如果设置, 每次写入前文件偏移量应设置为文件末尾。
- **O_CLOEXEC**: 如果设置, 新文件描述符的 `FD_CLOEXEC` 标志应被设置。
- **O_CLOFORK**: 如果设置, 新文件描述符的 `FD_CLOFORK` 标志应被设置。
- **O_CREAT**: 如果文件存在, 此标志无效, 除非下面 `O_EXCL` 中所述。否则, 如果未设置 `O_DIRECTORY`, 文件应作为常规文件创建; 文件的用户 ID 应设置为进程的有效用户 ID; 文件的组 ID 应设置为文件父目录的组 ID 或进程的有效组 ID; 文件模式的访问权限位 (参见 `<sys/stat.h>`) 应设置为 `oflag` 参数后的参数值, 类型为 `mode_t`, 修改如下: 对文件模式位和进程文件模式创建掩码的补码中的相应位执行按位与操作。
- **O_DIRECTORY**: 如果 `path` 解析为非目录文件, 失败并将 `errno` 设置为 `[ENOTDIR]`。
- **O_DSYNC** [SIO]: 文件描述符上的写 I/O 操作应按同步 I/O 数据完整性完成的要求完成。
- **O_EXCL**: 如果设置了 `O_CREAT` 和 `O_EXCL`, 且文件存在, `open()` 应失败。对文件存在性的检查以及文件不存在时创建文件的操作, 相对于其他在相同目录中执行 `open()` 并命名相同文件名且设置了 `O_EXCL` 和 `O_CREAT` 的线程应是原子的。如果设置了 `O_EXCL` 和 `O_CREAT`, 且 `path` 命名一个符号链接, `open()` 应失败并将 `errno` 设置为 `[EEXIST]`, 无论符号链接的内容如何。
- **O_NOCTTY**: 如果设置且 `path` 标识终端设备, `open()` 不应导致终端设备成为进程的控制终端。
- **O_NOFOLLOW**: 如果 `path` 命名符号链接, 失败并将 `errno` 设置为 `[ELOOP]`。
- **O_NONBLOCK**: 当使用 `O_RDONLY` 或 `O_WRONLY` 打开 FIFO 时:
 - 如果设置了 `O_NONBLOCK`, 只读 `open()` 应立即返回。只写 `open()` 如果当前没有进程为读取而打开文件, 应返回错误。

- 如果清除了 `O_NONBLOCK`，只读 `open()` 应阻塞调用线程，直到某个线程为写入而打开文件。只写 `open()` 应阻塞调用线程，直到某个线程为读取而打开文件。

当打开支持非阻塞打开的块特殊文件或字符特殊文件时：

- 如果设置了 `O_NONBLOCK`，`open()` 函数应立即返回，不等待设备就绪或可用。
- 如果清除了 `O_NONBLOCK`，`open()` 函数应阻塞调用线程，直到设备就绪或可用后再返回。
- `O_RSYNC` [SIO]: 文件描述符上的读 I/O 操作应完成在与 `O_DSYNC` 和 `O_SYNC` 标志指定的相同完整性级别。
- `O_SYNC` [XSI|SIO]: 文件描述符上的写 I/O 操作应按同步 I/O 文件完整性完成的要求完成。即使不支持同步输入和输出选项，也应支持常规文件的 `O_SYNC` 标志。
- `O_TRUNC`: 如果文件存在且是常规文件，且文件成功以 `O_RDWR` 或 `O_WRONLY` 方式打开，其长度应截断为 0，模式和所有者应保持不变。
- `O_TTY_INIT`: 如果 `path` 标识非伪终端的终端设备，设备尚未在任何进程中打开，且在 `oflag` 中设置了 `O_TTY_INIT` 或 `O_TTY_INIT` 值为零，`open()` 应将任何非标准 `termios` 结构终端参数设置为提供符合性行为的状态，并将与终端关联的 `winsize` 结构初始化为适当的默认设置。

`openat()` 函数应等效于 `open()` 函数，但 `path` 指定相对路径的情况除外。在这种情况下，要打开的文件是相对于与文件描述符 `fd` 关联的目录确定的，而不是相对于当前工作目录确定。

如果 `openat()` 在 `fd` 参数中传递特殊值 `AT_FDCWD`，应使用当前工作目录，行为应与调用 `open()` 完全相同。

返回值

成功完成后，这些函数应打开文件并返回一个代表文件描述符的非负整数。否则，这些函数应返回 -1 并设置 `errno` 以指示错误。如果返回 -1，不应创建或修改任何文件。

错误

在以下情况下，这些函数应失败：

- **[EACCES]**: 搜索路径前缀某个组件的权限被拒绝, 或文件存在但 `oflag` 指定的权限被拒绝, 或文件不存在且要创建文件的父目录的写权限被拒绝, 或指定了 `O_TRUNC` 但写权限被拒绝。
- **[EEXIST]**: 设置了 `O_CREAT` 和 `O_EXCL`, 且指定文件存在。
- **[EILSEQ]**: 指定了 `O_CREAT`, 文件不存在, 且 `path` 的最后一个路径组件不是可移植文件名且无法在目标目录中创建。
- **[EINTR]**: 在 `open()` 期间捕获到信号。
- **[EINVAL]**: `path` 参数命名一个 FIFO, 指定了 `O_RDWR`, 且实现认为这是一个错误; 或指定了同步 I/O 标志但实现不支持文件的同步 I/O。
- **[EISDIR]**: 指定文件是目录且 `oflag` 包含 `O_WRONLY` 或 `O_RDWR`, 或包含不带 `O_DIRECTORY` 的 `O_CREAT`, 或在 `O_EXEC` 与 `O_SEARCH` 不是同一个值时包含 `O_EXEC`。
- **[ELOOP]**: 在解析 `path` 参数过程中遇到符号链接循环, 或指定了 `O_NOFOLLOW` 且 `path` 参数命名符号链接。
- **[EMFILE]**: 进程可用的所有文件描述符当前都已打开。
- **[ENAMETOOLONG]**: 路径名某个组件的长度超过 `{NAME_MAX}`。
- **[ENFILE]**: 系统当前已打开的文件数量达到最大允许数量。
- **[ENOENT]**: 未设置 `O_CREAT` 且 `path` 的某个组件不命名现有文件, 或设置了 `O_CREAT` 且 `path` 路径前缀的某个组件不命名现有文件, 或 `path` 指向空字符串。
- **[ENOENT]** 或 **[ENOTDIR]**: 设置了 `O_CREAT`, 且 `path` 参数包含至少一个非 `<slash>` 字符并以一个或多个尾部 `<slash>` 字符结尾。
- **[ENOSPC]**: 将包含新文件的目录或文件系统无法扩展, 文件不存在, 且指定了 `O_CREAT`。
- **[ENOTDIR]**: 路径前缀的某个组件命名一个现有文件, 该文件既不是目录也不是指向目录的符号链接; 或未指定 `O_CREAT` 和 `O_EXCL`, `path` 参数包含至少一个非 `<slash>` 字符并以一个或多个尾部 `<slash>` 字符结尾; 或指定了 `O_DIRECTORY` 且 `path` 参数命名非目录文件。
- **[ENXIO]**: 设置了 `O_NONBLOCK`, 指定文件是 FIFO, 设置了 `O_WRONLY`, 且没有进程为读取而打开文件。
- **[ENXIO]**: 指定文件是字符特殊文件或块特殊文件, 与此特殊文件关联的设备不存在。

- **[EOVERFLOW]**: 指定文件是常规文件且文件大小无法在 `off_t` 类型的对象中正确表示。
- **[EROFS]**: 指定文件驻留在只读文件系统上，且在 `oflag` 参数中设置了 `O_WRONLY`、`O_RDWR`、`O_CREAT` (如果文件不存在) 或 `O_TRUNC`。

在以下情况下，`openat()` 函数应失败：

- **[EACCES]**: 与 `fd` 关联的打开文件描述的访问模式不是 `O_SEARCH`，且 `fd` 底层目录的权限不允许目录搜索。
- **[EBADF]**: `path` 参数未指定绝对路径且 `fd` 参数既不是 `AT_FDCWD` 也不是为读取或搜索打开的有效文件描述符。
- **[ENOTDIR]**: `path` 参数不是绝对路径且 `fd` 是与非目录文件关联的文件描述符。

在以下情况下，这些函数可能失败：

- **[EAGAIN]** [XSI]: `path` 参数命名被锁定的伪终端设备的从属端。
- **[EINVAL]**: `oflag` 参数的值无效。
- **[ELOOP]**: 在解析 `path` 参数过程中遇到超过 `{SYMLINK_MAX}` 个符号链接。
- **[ENAMETOOLONG]**: 路径名长度超过 `{PATH_MAX}`，或符号链接的路径名解析产生长度超过 `{PATH_MAX}` 的中间结果。
- **[EOPNOTSUPP]**: `path` 参数命名套接字。
- **[ETXTBSY]**: 文件是正在执行的纯过程（共享文本）文件，且 `oflag` 是 `O_WRONLY` 或 `O_RDWR`。

示例

按所有者权限写入打开文件

```
#include <fcntl.h>
...
int fd;
mode_t mode = S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH;
char *pathname = "/tmp/file";
...
fd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC, mode);
...
```

使用存在性检查打开文件

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd; /* open() 调用返回的文件描述符整数。 */
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n"
    exit(1);
}
...

```

为写入打开文件

```
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>

#define LOCKFILE "/etc/ptmp"
...
int pfd;
char pathname[PATH_MAX+1];
...
if ((pfd = open(pathname, O_WRONLY | O_CREAT | O_TRUNC,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    perror("Cannot open output file\n");
    exit(1);
}
...
```

应用程序用法

POSIX.1-2024 不要求终端参数在首次打开时自动设置为任何状态，也不要求在最后一次关闭后重置。为确保设备设置为符合性的初始状态，首次打开终端（伪终端除外）的应用程序应使用 `O_TTY_INIT` 标志进行操作。

除非在 POSIX.1-2024 本卷中另有规定, `oflag` 中允许的标志不是互斥的, 可以同时使用任意数量的标志。

`open()` 的 `O_CLOEXEC` 和 `O_CLOFORK` 标志对于避免多线程应用程序中的数据竞争是必要的。

另请参见

`chmod()`、`close()`、`creat()`、`dirfd()`、`dup()`、`exec()`、
`fcntl()`、`fdopendir()`、`link()`、`lseek()`、`mkdtemp()`、
`mknod()`、`read()`、`symlink()`、`umask()`、`unlockpt()`、
`write()`

XBD 11. General Terminal Interface、`<fcntl.h>`、`<sys/stat.h>`、
`<sys/types.h>`

1.102. pause

概要

```
#include <unistd.h>

int pause(void);
```

描述

`pause()` 函数应暂停调用线程，直到传递一个其动作是执行信号捕获函数或终止进程的信号。

如果动作是终止进程，`pause()` 不应返回。

如果动作是执行信号捕获函数，`pause()` 应在信号捕获函数返回后返回。

返回值

由于 `pause()` 无限期暂停线程执行，除非被信号中断，因此没有成功完成的返回值。应返回值 -1 并设置 `errno` 以指示错误。

错误

`pause()` 函数在以下情况下应失败：

- **[EINTR]**

调用进程捕获了一个信号，并且控制从信号捕获函数返回。

以下各节为参考信息。

示例

无。

应用程序使用

`pause()` 的许多常见用法存在时间窗口问题。这种情况涉及检查与信号相关的条件，如果信号尚未发生，则调用 `pause()`。当信号在检查和调用 `pause()` 之间发生时，进程通常会无限期阻塞。可以使用 `sigprocmask()` 和 `sigsuspend()` 函数来避免这类问题。

基本原理

无。

未来方向

无。

参见

- `sigsuspend()`
- XBD `<unistd.h>`

变更历史

首次发布于第1版。派生自 SVID 第1版。

第5版

为与 POSIX 线程扩展对齐，更新了描述部分。

第6版

添加了应用程序使用部分。

1.103. perror

SYNOPSIS

```
#include <stdio.h>

void perror(const char *s);
```

DESCRIPTION

`perror()` 函数应将通过符号 `errno` 访问的错误号映射到与语言相关的错误消息，该消息应按以下格式写入标准错误流：

- 首先（如果 `s` 不是空指针且 `s` 指向的字符不是空字节），写入 `s` 指向的字符串，后跟一个 `<冒号>` 和一个 `<空格>`。
- 然后是错误消息字符串，后跟一个 `<换行符>`。

错误消息字符串的内容应与 `strerror()` 函数以 `errno` 为参数返回的内容相同。

`perror()` 函数应在成功完成与 `exit()`、`abort()` 或在 `stderr` 上完成 `fflush()` 或 `fclose()` 之间的某个时间，标记与标准错误流关联的文件的数据修改和文件状态更改的最后时间戳以供更新。

`perror()` 函数不应改变标准错误流的方向。

出错时，`perror()` 应为 `stderr` 指向的流设置错误指示器，并应设置 `errno` 以指示错误。

由于不返回任何值，希望检查错误情况的应用程序应在调用 `perror()` 之前调用 `clearerr(stderr)`，然后如果 `ferror(stderr)` 返回非零值，`errno` 的值指示发生了哪个错误。

RETURN VALUE

`perror()` 函数不应返回任何值。

ERRORS

参考 `fputc()`。

EXAMPLES

为函数打印错误消息

以下示例将 `bufptr` 替换为具有必要大小的缓冲区。如果发生错误，`perror()` 函数打印一条消息且程序退出。

```
#include <stdio.h>
#include <stdlib.h>
...
char *bufptr;
size_t szbuf;
...
if ((bufptr = malloc(szbuf)) == NULL) {
    perror("malloc");
    exit(2);
}
...
```

APPLICATION USAGE

应用程序编写者可能更喜欢使用替代接口而不是 `perror()`，例如将 `strerror_r()` 与 `fprintf()` 结合使用。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `fprintf()`
- `fputc()`
- `psiginfo()`
- `strerror()`
- `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5: 向 DESCRIPTION 添加了一段，指出 `perror()` 不改变标准错误流的方向。

Issue 6: 标记了超出 ISO C 标准的扩展。

Issue 7: 应用了 SD5-XSH-ERN-95。进行了与支持细粒度时间戳相关的更改。应用了 POSIX.1-2008、Technical Corrigendum 1、XSH/TC1-2008/0429 [389,401]、XSH/TC1-2008/0430 [389] 和 XSH/TC1-2008/0431 [389,401]。

1.104. printf

概要

```
#include <stdio.h>

int asprintf(char **restrict ptr, const char *restrict format,
int dprintf(int fildes, const char *restrict format, ...);
int fprintf(FILE *restrict stream, const char *restrict format,
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
            const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...)
```

描述

除了 asprintf()、dprintf() 以及传入空宽字符时 %lc 转换的行为外，本参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何其他冲突都是无意的。POSIX.1-2024 的这一卷在涉及传入空宽字符时的 %lc 转换方面，遵从 ISO C 标准处理所有 fprintf()、printf()、snprintf() 和 sprintf() 功能。

fprintf() 函数应将输出放置在指定的输出流上。printf() 函数应将输出放置在标准输出流 stdout 上。sprintf() 函数应将输出后跟空字节 '\0' 放置在从 *s 开始的连续字节中；用户有责任确保有足够的空间可用。

asprintf() 函数应等同于 sprintf()，不同之处在于输出字符串应写入动态分配的内存，该内存长度上足以容纳生成的字符串（包括终止空字节），分配方式如同调用 malloc()。如果对 asprintf() 的调用成功，则此动态分配字符串的地址应存储在 ptr 引用的位置。

dprintf() 函数应等同于 fprintf() 函数，不同之处在于 dprintf() 应将输出写入与 fildes 参数指定的文件描述符关联的文件，而不是将输出放置在流上。

snprintf() 函数应等同于 sprintf()，增加了 n 参数，该参数限制写入 s 引用的缓冲区的字节数。如果 n 为零，则不应写入任何内容，s 可能为空指针。否则，超过第 n-1 个的输出字节将被丢弃而不是写入数组，并且在实际写入数组的字节末尾写入一个空字节。

如果由于调用 sprintf() 或 snprintf() 导致在重叠的对象之间进行复制，则结果未定义。

这些函数中的每个都在格式的控制下转换、格式化并打印其参数。应用程序应确保格式是一个字符串，如果存在初始转换状态，则在该状态下开始和结束。格式由零个或多个指令组成：普通字符（简单地复制到输出流）和转换规范（每个都将导致获取零个或多个参数）。如果格式没有足够的参数，则结果未定义。如果格式用尽而参数仍有剩余，则多余的参数将被求值，但除此之外将被忽略。

转换可以应用于参数列表中格式之后的第 n 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 %（见下文）被序列 "%n\$" 替换，其中 n 是范围 [1,{NL_ARGMAX}] 内的十进制整数，给出参数在参数列表中的位置。此功能提供了定义以适合特定语言的顺序选择参数的格式字符串的功能（请参阅示例部分）。

格式可以包含编号参数转换规范（即由 "%n\$" 引入并可选包含字段宽度和精度的 "*m\$" 形式），或未编号参数转换规范（即由 % 字符引入并可选包含字段宽度和精度的 * 形式），但不能两者兼有。唯一的例外是 %% 可以与 "%n\$" 形式混合使用。在格式字符串中混合编号和未编号参数规范的结果未定义。使用编号参数规范时，指定第 N 个参数要求所有前导参数（从第一个到第 (N-1) 个）都在格式字符串中指定。

在包含 "%n\$" 形式转换规范的格式字符串中，参数列表中的编号参数可以从格式字符串中根据需要引用多次。

在包含 % 形式转换规范的格式字符串中，每个转换规范使用参数列表中的第一个未使用参数。

所有形式的 `fprintf()` 函数都允许在输出字符串中插入与语言相关的基数字符。基数字符在当前语言环境（类别 `LC_NUMERIC`）中定义。在 POSIX 语言环境或未定义基数字符的语言环境中，基数字符应默认为句点 ('.')。

每个转换规范由 '%' 字符或字符序列 "%n\$" 引入，其后依次出现以下内容：

1. 零个或多个标志（以任意顺序），它们修改转换规范的含义。
2. 可选的最小字段宽度。如果转换后的值具有比字段宽度更少的字节，则默认应在左侧用空格字符填充；如果给出了左调整标志 ('-')（如下所述），则应在右侧填充。字段宽度采用星号 ('*') 的形式，或在由 "%n\$" 引入的转换规范中采用下面描述的 "m\$" 字符串，或十进制整数的形式。
3. 可选的精度，它给出 d、i、o、u、x 和 X 转换说明符要出现的最小数字位数；a、A、e、E、f 和 F 转换说明符要在基数字符后出现的数字位数；g 和 G 转换说明符的最大有效数字位数；或要从 s 和 S 转换说明符打印的字符串的最大字节数。精度采用句点 ('.') 后跟星号 ('*') 的形式，或在由 "%n\$" 引入的转换规范中采用下面描述的 "m\$" 字符串，或可选的十进制数字字符串的形式，其中空数字字符串被视为零。如果精度与任何其他转换说明符一起出现，则行为未定义。

4. 可选的长度修饰符，指定参数的大小。

5. 转换说明符字符，指示要应用的转换类型。

字段宽度、精度或两者都可以用星号 ('*) 表示。在这种情况下，int 类型的参数提供字段宽度或精度。应用程序应确保指定字段宽度、精度或两者的参数按该顺序出现在要转换的参数（如果有）之前。负的字段宽度被视为 '-' 标志后跟正的字段宽度。负的精度被视为精度被省略。在包含由 "%n\$" 引入的转换规范的格式字符串中，除了用十进制数字字符串表示外，字段宽度还可以用序列 "m\$" 表示，精度用序列 ".m\$" 表示，其中 m 是范围 [1,{NL_ARGMAX}] 内的十进制整数，给出包含字段宽度或精度的整数参数在参数列表中的位置（在格式参数之后），例如：

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

标志字符

标志字符及其含义是：

' (撇号) - 十进制转换 (%i、%d、%u、%f、%F、%g 或 %G) 结果的整数部分应使用千位分组字符格式化。对于其他转换，行为未定义。使用非货币分组字符。

-- 转换结果应在字段内左对齐。如果未指定此标志，则转换右对齐。

+ - 有符号转换的结果应始终以符号 ('+' 或 '-') 开头。如果未指定此标志，则仅当转换负值时才以符号开头。

(空格) - 如果有符号转换的第一个字符不是符号，或者有符号转换不产生任何字符，则应在结果前加空格。这意味着如果空格和 '+' 标志同时出现，则忽略空格标志。

- 指定值要转换为替代形式。对于 o 转换，它应增加精度，当且仅当需要时，以强制结果的第一个数字为零（如果值和精度都为 0，则打印单个 0）。对于 x 或 X 转换说明符，非零结果应前缀 0x（或 0X）。对于 a、A、e、E、f、F、g 和 G 转换说明符，结果应始终包含基数字符，即使基数字符后没有数字。没有此标志时，基数字符仅当后有数字时才出现在这些转换的结果中。对于 g 和 G 转换说明符，结果末尾的零不应像通常那样被删除。对于其他转换说明符，行为未定义。

0 - 对于 d、i、o、u、x、X、a、A、e、E、f、F、g 和 G 转换说明符，使用前导零（在任何符号或基数指示之后）填充到字段宽度，而不是执行空格填充，除非转换无穷大或 NaN。如果 '0' 和 '-' 标志同时出现，则忽略 '0' 标志。对于 d、i、o、u、x 和 X 转换说明符，如果指定了精度，则忽略 '0' 标志。如果 '0' 和撇

号标志同时出现，则在零填充之前插入分组字符。对于其他转换，行为未定义。

长度修饰符

长度修饰符及其含义是：

hh - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 signed char 或 unsigned char 参数（参数将根据整数提升被提升，但其值应在打印前转换为 signed char 或 unsigned char）；或指定后续的 n 转换说明符应用于指向 signed char 参数的指针。

h - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 short 或 unsigned short 参数（参数将根据整数提升被提升，但其值应在打印前转换为 short 或 unsigned short）；或指定后续的 n 转换说明符应用于指向 short 参数的指针。

l (小写字母 l) - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 long 或 unsigned long 参数；指定后续的 n 转换说明符应用于指向 long 参数的指针；指定后续的 c 转换说明符应用于 wint_t 参数；指定后续的 s 转换说明符应用于指向 wchar_t 参数的指针；或对后续的 a、A、e、E、f、F、g 或 G 转换说明符没有影响。

ll (小写字母 l-l) - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 long long 或 unsigned long long 参数；或指定后续的 n 转换说明符应用于指向 long long 参数的指针。

j - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 intmax_t 或 uintmax_t 参数；或指定后续的 n 转换说明符应用于指向 intmax_t 参数的指针。

z - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 size_t 或相应的有符号整数类型参数；或指定后续的 n 转换说明符应用于指向与 size_t 参数对应的有符号整数类型的指针。

t - 指定后续的 d、i、o、u、x 或 X 转换说明符应用于 ptrdiff_t 或相应的无符号类型参数；或指定后续的 n 转换说明符应用于指向 ptrdiff_t 参数的指针。

L - 指定后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于 long double 参数。

如果长度修饰符与除上述指定外的任何转换说明符一起出现，则行为未定义。

转换说明符

转换说明符及其含义是：

d, i - int 参数应转换为 "[-]dddd" 形式的有符号十进制数。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

o - 无符号参数应转换为 "dddd" 形式的无符号八进制数。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

u - 无符号参数应转换为 "dddd" 形式的无符号十进制数。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

x - 无符号参数应转换为 "dddd" 形式的无符号十六进制数；使用字母 "abcdef"。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

X - 等同于 x 转换说明符，但使用字母 "ABCDEF" 而不是 "abcdef"。

f, F - double 参数应转换为 "[-]ddd.ddd" 形式的十进制表示法，其中基数字符后的数位数等于精度规范。如果精度缺失，则视为 6；如果精度显式为零且没有 '#' 标志，则不应出现基数字符。如果出现基数字符，则它前面至少出现一个数字。低位数字应以实现定义的方式四舍五入。

表示无穷大的 double 参数应转换为 "[-]inf" 或 "[-]infinity" 之一的形式；使用哪种形式是实现定义的。表示 NaN 的 double 参数应转换为 "[-]nan(n-char-sequence)" 或 "[-]nan" 之一的形式；使用哪种形式以及任何 n-char-sequence 的含义是实现定义的。F 转换说明符产生 "INF"、"INFINITY" 或 "NAN" 而不是 "inf"、"infinity" 或 "nan"。

e, E - double 参数应转换为 "[-]d.ddd_e±dd" 形式，其中基数字符前有一个数字（如果参数非零，则该数字非零），其后的数位数等于精度；如果精度缺失，则视为 6；如果精度为零且没有 '#' 标志，则不应出现基数字符。低位数字应以实现定义的方式四舍五入。E 转换说明符应产生用 'E' 代替 'e' 引入指数的数字。指数应始终包含至少两位数字。如果值为零，则指数为零。

表示无穷大或 NaN 的 double 参数应以 f 或 F 转换说明符的形式转换。

g, G - 表示浮点数的 double 参数应根据转换的值和精度以 f 或 e 形式（或在 G 转换说明符的情况下以 F 或 E 形式）转换。设 P 等于精度（如果非零），如果精度省略则为 6，如果精度为零则为 1。那么，如果使用 E 形式的转换指数为 X：

- 如果 $P > X \geq -4$ ，则转换使用 f（或 F）形式和精度 $P-(X+1)$ 。
- 否则，转换使用 e（或 E）形式和精度 P-1。

最后，除非使用 '#' 标志，否则结果的小数部分应删除任何尾随零，如果没有剩余的小数部分，则删除小数点字符。

表示无穷大或 NaN 的 double 参数应以 f 或 F 转换说明符的形式转换。

a, A - 表示浮点数的 double 参数应转换为 "[-]0x_h_.hhhh_p±d" 形式，其中小数点字符前有一个十六进制数字（如果参数是归一化浮点数，则该数字非零，否则未指定），其后的十六进制数位数等于精度；如果精度缺失且 FLT_RADIX 是 2 的幂，则精度应足以精确表示该值；如果精度缺失且 FLT_RADIX 不是 2 的幂，则精度应足以区分 double 类型的值，但可以省略尾随零；如果精度为零且未指定 '#' 标志，则不应出现小数点字符。对于 a 转换使用字母 "abcdef"，对于 A 转换使用字母 "ABCDEF"。A 转换说明符产生用 'X' 和 'P' 代替 'x' 和 'p' 的数字。指数应始终包含至少一个数字，并且只包含表示 2 的十进制指数所需的更多数字。如果值为零，则指数为零。

表示无穷大或 NaN 的 double 参数应以 f 或 F 转换说明符的形式转换。

c - int 参数应转换为 unsigned char，并将结果字节写入。

如果存在 l（小写字母 l）限定符，则 wint_t 参数应转换为多字节序列，如同调用 wcrtomb() 并带有指向至少 MB_CUR_MAX 字节存储的指针、转换为 wchar_t 的 wint_t 参数和初始转换状态，并写入结果字节。

s - 参数应是指向 char 数组的指针。应写入数组中的字节直到（但不包括）任何终止空字节。如果指定了精度，则写入的字节数不应超过该数量。如果未指定精度或精度大于数组大小，则应用程序应确保数组包含空字节。

如果存在 l（小写字母 l）限定符，则参数应是指向 wchar_t 类型数组的指针。数组中的宽字符应转换为字符（每个如同调用 wcrtomb() 函数，转换状态由在第一个宽字符转换前初始化为零的 mbstate_t 对象描述）直到包括终止空宽字符。应写入结果字符直到（但不包括）终止空字符（字节）。如果未指定精度，则应用程序应确保数组包含空宽字符。如果指定了精度，则写入的字符（字节）数不应超过该数量（包括任何移位序列，如果有），并且如果为等于精度给出的字符序列长度，函数需要访问数组末尾之后的一个宽字符，则数组应包含空宽字符。在任何情况下都不应写入部分字符。

p - 参数应是指向 void 的指针。指针的值以实现定义的方式转换为可打印字符序列。

n - 参数应是指向整数的指针，其中写入此次调用 fprintf() 函数之一到目前为止写入输出的字节数。不转换任何参数。

C - 等同于 lc。

S - 等同于 ls。

% - 写入 '%' 字符；不应转换任何参数。应用程序应确保完整的转换规范为 %%。

如果转换规范与上述形式之一不匹配，则行为未定义。如果任何参数不是相应转换规范的正确类型，则行为未定义。

在任何情况下，不存在或过小的字段宽度都不会导致字段截断；如果转换结果宽于字段宽度，则应扩展字段以包含转换结果。由 `fprintf()` 和 `printf()` 生成的字符被打印如同调用了 `fputc()`。

对于 a 和 A 转换说明符，如果 `FLT_RADIX` 是 2 的幂，则值应正确四舍五入为具有给定精度的十六进制浮点数。

对于 a 和 A 转换，如果 `FLT_RADIX` 不是 2 的幂且结果不能以给定精度精确表示，则结果应为给定精度的十六进制浮点样式中的两个相邻数字之一，附加要求是误差对于当前舍入方向应具有正确的符号。

对于 e、E、f、F、g 和 G 转换说明符，如果有效十进制数字的数量最多为 `DECIMAL_DIG`，则结果应正确四舍五入。如果有效十进制数字的数量超过 `DECIMAL_DIG` 但源值可以用 `DECIMAL_DIG` 位数字精确表示，则结果应为带有尾随零的精确表示。否则，源值由两个相邻的十进制字符串 `L < U` 限定，两者都具有 `DECIMAL_DIG` 位有效数字；结果十进制字符串 `D` 的值应满足 `L <= D <= U`，附加要求是误差对于当前舍入方向应具有正确的符号。

文件的最后数据修改时间和最后文件状态更改时间应标记为更新：

1. 在调用成功执行 `fprintf()` 或 `printf()` 与下一次在同一流上成功完成调用 `fflush()` 或 `fclose()` 或调用 `exit()` 或 `abort()` 之间
2. 成功完成调用 `dprintf()` 时

返回值

成功完成时，`dprintf()`、`fprintf()` 和 `printf()` 函数应返回传输的字节数。

成功完成时，`asprintf()` 函数应返回写入存储在 `ptr` 引用位置的分配字符串的字节数，不包括终止空字节。

成功完成时，`sprintf()` 函数应返回写入 `s` 的字节数，不包括终止空字节。

成功完成时，`snprintf()` 函数应返回如果 `n` 足够大则将写入 `s` 的字节数，不包括终止空字节。

如果遇到错误，这些函数应返回负值并设置 `errno` 以指示错误。对于 `asprintf()`，如果无法分配内存，或者发生其他错误，则函数应返回负值，`ptr` 引用的位置的内容未定义，但不应引用已分配的内存。

如果在调用 `snprintf()` 时 `n` 的值为零，则不应写入任何内容，应返回如果 `n` 足够大则将写入的字节数（不包括终止空字节），且 `s` 可能为空指针。

错误

有关 `dprintf()`、`fprintf()` 和 `printf()` 失败和可能失败的条件，请参考 `fputc()` 或 `fputwc()`。

此外，所有形式的 `fprintf()` 在以下情况下应失败：

- **[EILSEQ]** - 检测到与有效字符不对应的宽字符代码。
- **[EOVERFLOW]** - 要返回的值大于 `{INT_MAX}`。

`asprintf()` 函数在以下情况下应失败：

- **[ENOMEM]** - 可用存储空间不足。

`dprintf()` 函数在以下情况下可能失败：

- **[EBADF]** - `fildes` 参数不是有效的文件描述符。

`dprintf()`、`fprintf()` 和 `printf()` 函数在以下情况下可能失败：

- **[ENOMEM]** - 可用存储空间不足。

示例

打印语言无关的日期和时间

以下语句可用于使用语言无关的格式打印日期和时间：

```
printf(format, weekday, month, day, hour, min);
```

对于美国用法，`format` 可以是指向以下字符串的指针：

```
"%s, %s %d, %d:%.2d\n"
```

此示例将产生以下消息：

```
Sunday, July 3, 10:02
```

对于德国用法，`format` 可以是指向以下字符串的指针：

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

此 `format` 定义将产生以下消息：

```
Sonntag, 3. Juli, 10:02
```

打印文件信息

以下示例打印目录中特定文件的类型、权限和链接数信息。

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
```

打印本地化日期字符串

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT),
printf(" %s %s\n", datestring, dp->d_name);
```

打印错误信息

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"
...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n"
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
```

打印使用信息

```
#include <stdio.h>
#include <stdlib.h>
...
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options);
}
```

格式化十进制字符串

```
#include <stdio.h>
...
```

```
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

创建路径名

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
...
char *pathname;
struct passwd *pw;
size_t len;
...
// pid_t 所需的位数是位数乘以
// log2(10) = 约等于 10/33
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +
      sizeof ".out";
pathname = malloc(len);
if (pathname != NULL)
{
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,
             (intmax_t)getpid());
    ...
}
```

报告事件

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
...
while (!event_complete) {
...
    if (pause() != -1 || errno != EINTR)
```

```
        fprintf(stderr, "pause: unknown error: %s\n", strerror(  
    }
```

打印货币信息

```
#include <monetary.h>  
#include <stdio.h>  
...  
struct tblfmt {  
    char *format;  
    char *description;  
};  
  
struct tblfmt table[] = {  
    { "%n", "default formatting" },  
    { "%11n", "right align within an 11 character field" },  
    { "%#5n", "aligned columns for values up to 99999" },  
    { "%=#5n", "specify a fill character" },  
    { "%=0#5n", "fill characters do not use grouping" },  
    { "%^#5n", "disable the grouping separator" },  
    { "%^#5.0n", "round off to whole units" },  
    { "%^#5.4n", "increase the precision" },  
    { "%(#5n", "use an alternative pos/neg style" },  
    { "%!(#5n", "disable the currency symbol" },  
};  
...  
float input[3];  
int i, j;  
char convbuf[100];  
...  
strfmmon(convbuf, sizeof(convbuf), table[i].format, input[j]);  
  
if (j == 0) {  
    printf("%s %s %s\n", table[i].format,  
          convbuf, table[i].description);  
}  
else {  
    printf(" %s\n", convbuf);  
}
```

打印宽字符

以下示例打印一系列宽字符。假设 "L @" 扩展为三个字节：

```
wchar_t wz [3] = L"@@";           // 以零终止
wchar_t wn [3] = L"@@";           // 未终止

fprintf (stdout,"%ls", wz);     // 输出 6 字节
fprintf (stdout,"%ls", wn);     // 未定义, 因为 wn 没有终止符
fprintf (stdout,"%4ls", wz);    // 输出 3 字节
fprintf (stdout,"%4ls", wn);    // 输出 3 字节; 不需要终止符
fprintf (stdout,"%9ls", wz);    // 输出 6 字节
fprintf (stdout,"%9ls", wn);    // 输出 9 字节; 不需要终止符
fprintf (stdout,"%10ls", wz);   // 输出 6 字节
fprintf (stdout,"%10ls", wn);   // 未定义, 因为 wn 没有终止符
```

在示例的最后一行, 处理三个字符后, 已输出九个字节。然后必须检查第四个字符以确定它转换为一个还是多个字节。如果它转换为多个字节, 则输出只有九个字节。由于数组中没有第四个字符, 因此行为未定义。

应用程序用法

如果调用 `fprintf()` 的应用程序具有任何 `wint_t` 或 `wchar_t` 类型的对象, 则它还必须包含头文件以定义这些对象。

成功调用 `asprintf()` 分配的空间应随后通过调用 `free()` 释放。

基本原理

如果实现检测到格式没有足够的参数, 建议函数应失败并报告 `[EINVAL]` 错误。

为 `%lc` 转换指定的行为与 ISO C 标准中的规范略有不同, 因为打印空宽字符产生空字节而不是 0 字节输出, 这是严格按照 ISO C 标准指示表现得像将 `%ls` 说明符应用于第一个元素是空宽字符的 `wchar_t` 数组所要求的。为每个可能的宽字符 (包括空字符) 要求多字节输出符合历史实践, 并与 `fprintf()` 中的 `%c` 以及 `fwprintf()` 中的 `%c` 和 `%lc` 保持一致。预计 ISO C 标准的未来版本将更改以匹配此处指定的行为。

未来方向

无。

另请参阅

- 2.5 标准 I/O 流
- fputc()
- fscanf()
- setlocale()
- strfmon()
- strlcat()
- wcrtomb()
- wcslcat()

XBD:

- 7. 语言环境
-
-
-

更改历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

与 ISO/IEC 9899:1990/Amendment 1:1995 (E) 对齐。具体来说，l (小写字母 l) 限定符现在可以与 c 和 s 转换说明符一起使用。

snprintf() 函数是 Issue 5 中的新增内容。

Issue 6

超出 ISO C 标准的扩展被标记。

规范性文本更新，避免对应用程序要求使用术语“必须”。

为与 ISO/IEC 9899:1999 标准对齐进行以下更改：

- fprintf()、printf()、snprintf() 和 sprintf() 的原型更新，并从 snprintf() 中移除了 XSI 底纹。

- `snprintf()` 的描述与 ISO C 标准对齐。请注意，这取代了 The Open Group Base Resolution bwg98-006 中的 `snprintf()` 描述，该描述改变了 Issue 5 的行为。
- 描述部分更新。

描述部分更新为一致使用"转换说明符"和"转换规范"术语。

整合 ISO/IEC 9899:1999 标准，技术勘误 1。

添加打印宽字符的示例。

Issue 7

应用 Austin Group 解释 1003.1-2001 #161，更新 0 标志的描述。

应用 Austin Group 解释 1003.1-2001 #170。

应用 ISO/IEC 9899:1999 标准，技术勘误 2 #68 (SD5-XSH-ERN-70)，修订 g 和 G 的描述。

应用 SD5-XSH-ERN-174。

从 The Open Group 技术标准 2006，扩展 API 集第 1 部分添加 `dprintf()` 函数。

与 `%n$` 形式转换规范和撇号标志相关的功能从 XSI 选项移至基础。

进行与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0163 [302]、XSH/TC1-2008/0164 [316]、XSH/TC1-2008/0165 [316]、XSH/TC1-2008/0166 [451,291] 和 XSH/TC1-2008/0167 [14]。

应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0126 [894]、XSH/TC2-2008/0127 [557] 和 XSH/TC2-2008/0128 [936]。

Issue 8

应用 Austin Group 缺陷 986，将 `strlcat()` 和 `wcslcat()` 添加到"另请参阅"部分。

应用 Austin Group 缺陷 1020，澄清 `snprintf()` 参数 n 限制写入 s 的字节数；它不一定与 s 的大小相同。

应用 Austin Group 缺陷 1021，将"返回值"部分中的"输出错误"更改为"错误"。

应用 Austin Group 缺陷 1137，澄清在转换规范中使用 "`%n$`" 和 "`*m$`"。

应用 Austin Group 缺陷 1205，更改 % 转换说明符的描述。

应用 Austin Group 缺陷 1219，移除 `snprintf()` 特有的 [EOVERFLOW] 错误。

应用 Austin Group 缺陷 1496，添加 `asprintf()` 函数。

应用 Austin Group 缺陷 1562，澄清应用程序有责任确保格式是一个字符串，如果存在初始转换状态，则在该状态下开始和结束。

应用 Austin Group 缺陷 1647，更改 c 转换说明符的描述，并更新本卷 POSIX.1-2024 遵从 ISO C 标准的陈述，以便在传入空宽字符时排除 %lc 转换。

1.105. pthread_atfork

概要

```
#include <pthread.h>

int pthread_atfork(void (*prepare)(void), void (*parent)(void),
```

描述

以下章节为说明性内容。

基本原理

在多线程程序中，`fork()` 的语义至少存在两个严重问题。一个问题与互斥锁保护的状态（例如内存）有关。考虑这样的情况：一个线程锁定了互斥锁且该互斥锁保护的状态处于不一致状态，而另一个线程调用 `fork()`。在子进程中，互斥锁处于锁定状态（由不存在的线程锁定，因此永远无法解锁）。让子进程简单地重新初始化互斥锁是不能令人满意的，因为这种方法没有解决如何纠正或处理子进程中的不一致状态的问题。

建议使用 `fork()` 的程序在子进程中紧接着调用 `exec` 函数，从而重置所有状态。在此期间，只有简短的异步信号安全库例程列表可以保证可用。

不幸的是，这个解决方案没有满足多线程库的需求。应用程序可能不知道正在使用多线程库，它们可以在 `fork()` 和 `exec` 调用之间随意调用任意数量的库例程，就像它们一直以来那样。实际上，它们可能是现有的单线程程序，因此不能期望它们遵守线程库施加的新限制。

另一方面，多线程库需要一种方法来在 `fork()` 期间保护其内部状态，以防它在子进程的后续执行中被重新进入。这个问题在多线程 I/O 库中尤为突出，这些库几乎肯定会在 `fork()` 和 `exec` 调用之间被调用以实现 I/O 重定向。解决方案可能需要在 `fork()` 期间锁定互斥锁变量，或者它可能需要在 `fork()` 处理完成后在子进程中简单地重置状态。

`pthread_atfork()` 函数旨在为多线程库提供一种保护自己免受调用 `fork()` 的无辜应用程序影响的方法，并为多线程应用程序提供一种标准机制，以保护自己免受库例程或应用程序本身中的 `fork()` 调用的影响。

预期的用法是：prepare 处理函数将获取所有互斥锁，其他两个 fork 处理函数将释放它们。

例如，应用程序可以提供一个 prepare 例程来获取库维护的必要互斥锁，并提供释放这些互斥锁的 child 和 parent 例程，从而确保子进程能够获得库状态的一致快照（并且不会留下任何孤立的互斥锁）。这在理论上是好的，但在现实中不实用。必须定位并锁定进程中的每一个互斥锁和锁。程序的每个组件，包括第三方组件，都必须参与，并且它们必须就谁负责哪个互斥锁或锁达成一致。这对于动态分配内存中的互斥锁和锁尤其成问题。实现内部的所有互斥锁和锁也必须被锁定。这可能会延迟调用 `fork()` 的线程很长时间甚至无限期延迟，因为这些同步对象的使用可能不受应用程序控制。这里要提到的最后一个问题是锁定流的问题。至少系统控制下的流（如 `stdin`、`stdout`、`stderr`）必须通过使用 `flockfile()` 锁定流来保护。但应用程序本身可能已经这样做了，可能在同一个调用 `fork()` 的线程中。在这种情况下，进程将发生死锁。

或者，一些库可能能够只提供一个 `child` 例程，将库中的互斥锁和所有相关状态重新初始化为某个已知值（例如，镜像最初执行时的状态）。但是，这种方法是不可能的，因为如果互斥锁或锁仍然被锁定，实现允许失败互斥锁和锁的 `*_init_()` 和 `*_destroy_()` 调用。在这种情况下，`child` 例程无法重新初始化互斥锁和锁。

当调用 `fork()` 时，只有调用线程在子进程中被复制。同步变量在子进程中保持与调用 `fork()` 时在父进程中相同的状态。因此，例如，互斥锁可能由在子进程中不再存在的线程持有，并且任何相关状态可能不一致。预期是父进程可以通过显式代码来避免这种情况，这些代码通过 `pthread_atfork()` 获取和释放对子进程关键的锁。此外，任何关键线程都需要在子进程中重新创建并重新初始化到适当状态（也通过 `pthread_atfork()`）。

更高级别的包可能在调用低级包之前对其自己的数据结构获取锁。在这种情况下，为 fork 处理函数调用指定的顺序允许一个简单的初始化规则来避免包死锁：包在为自己调用 `pthread_atfork()` 函数之前初始化它依赖的所有包。

如上所述，对于需要通过互斥锁和锁来保护非原子操作的功能，没有合适的解决方案。这就是为什么自 1996 年发布以来 POSIX.1 标准要求在多线程进程中 `fork()` 后的子进程只调用异步信号安全接口。

当调用 `pthread_atfork()` 为使用 `dlopen()` 加载的库中的函数注册时，会出现另一个问题。如果使用 `dlclose()` 卸载库，并且 `dlclose()` 的实现没有取消注册该函数，那么当 `fork()` 尝试调用它时，结果将是未定义行为。一些 `dlclose()` 的实现确实取消注册 `pthread_atfork()` 处理函数，但可移植应用程序不能依赖于此。该标准没有提供可移植的方法来取消注册通过 `pthread_atfork()` 安装为处理函数的函数。

另请参见

- [exec](#)
- [fork](#)
- [flockfile](#)
- [dlopen](#)
- [dlclose](#)

1.106. pthread_attr_destroy

概要

```
#include <pthread.h>

int pthread_attr_destroy(pthread_attr_t *attr);
int pthread_attr_init(pthread_attr_t *attr);
```

描述

`pthread_attr_destroy()` 函数应当销毁一个线程属性对象。实现可能导致 `pthread_attr_destroy()` 将 `attr` 设置为实现定义的无效值。已销毁的 `attr` 属性对象可以使用 `pthread_attr_init()` 重新初始化；在对象被销毁后其他方式引用该对象的结果是未定义的。

`pthread_attr_init()` 函数应当使用给定实现使用的所有单个属性的默认值来初始化线程属性对象 `attr`。

当被 `pthread_create()` 使用时，生成的属性对象（可能通过设置单个属性值进行修改）定义了所创建线程的属性。单个属性对象可以在多个同时调用 `pthread_create()` 中使用。如果调用 `pthread_attr_init()` 时指定了已经初始化的 `attr` 属性对象，则结果是未定义的。

如果 `pthread_attr_destroy()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，则行为是未定义的。

返回值

成功完成后，这些函数应返回零；否则，应返回错误号以指示错误。

错误

这些函数可能在以下情况下失败：

- **EINVAL** - `attr` 指定的值不引用已初始化的线程属性对象（对于 `pthread_attr_destroy()`）。

- **EBUSY** - `attr` 指定的值引用已初始化的线程属性对象（对于 `pthread_attr_init()`）。

这些函数不应返回 `[EINTR]` 错误码。

示例

未提供示例。

应用程序使用

为线程、互斥锁和条件变量提供属性对象，作为支持这些领域未来可能标准化的一种机制，而不需要改变函数本身。

属性对象为线程的可配置方面提供了清晰的隔离。例如，“栈大小”是线程的一个重要属性，但不能以可移植的方式表示。在移植线程程序时，通常需要调整栈大小。使用属性对象可以通过允许将更改隔离在单个位置来帮助，而不是分散在每次线程创建的实例中。

属性对象可用于设置具有相似属性的线程“类”；例如，“具有大栈和高优先级的线程”或“具有最小栈的线程”。这些类可以在单个位置定义，然后在需要创建线程的任何地方引用。对“类”决策的更改变得直接明了，不需要对每个 `pthread_create()` 调用进行详细分析。

属性对象被定义为不透明类型，以帮助扩展性。如果这些对象被指定为结构，添加新属性将强制在扩展属性对象时重新编译所有多线程程序；如果不同程序组件由不同供应商提供，这可能是不可能的。

此外，不透明属性对象提供了提高性能的机会。参数有效性可以在设置属性时检查一次，而不是每次创建线程时检查。实现通常需要缓存创建成本高昂的内核对象。不透明属性对象提供了检测缓存对象因属性更改而变为无效的高效机制。

由于赋值在给定不透明类型上不一定定义，实现定义的默认值无法以可移植方式定义。这个问题的解决方案是允许属性对象由属性对象初始化函数动态初始化，以便默认值可以由实现自动提供。

基本原理

提供了以下建议作为所提供属性的建议替代方案：

1. 保持将通过按位包含或标志形成的参数传递给初始化例程（`pthread_create()`、`pthread_mutex_init()`、

`pthread_cond_init()` 的样式。包含标志的参数应该是不透明类型以便扩展。如果参数中没有设置标志，则对象以默认特征创建。实现可以指定实现定义的标志值和相关行为。

2. 如果需要对互斥锁和条件变量进行进一步专门化，实现可以指定对 `pthread_mutex_t` 和 `pthread_cond_t` 对象（而不是对属性对象）进行操作的附加过程。

这种解决方案的困难是：

1. 如果必须使用显式编码的按位包含或操作将位设置到位向量属性对象中，则位掩码不是不透明的。如果选项集超过 `int`，应用程序程序员需要知道每个位的位置。如果位通过封装（即获取和设置函数）设置或读取，那么位掩码只是当前定义的属性对象的实现，不应暴露给程序员。
2. 许多属性不是布尔值或非常小的整数值。例如，调度策略可以放在3位或4位中，但优先级需要5位或更多，从而在具有16位整数的机器上占用可能的16位中至少8位。因此，位掩码只能合理地控制是否设置了特定属性，它不能作为值本身的存储库。值需要指定为函数参数（不可扩展）、或通过设置结构字段（非不透明）、或通过获取和设置函数（使位掩码成为属性对象的冗余添加）。

栈大小被定义为可选属性，因为栈的概念本质上是机器相关的。例如，一些实现可能无法更改栈的大小，而其他实现可能不需要，因为栈页可能是不连续的，可以根据需要分配和释放。

属性机制在设计上很大程度上考虑了扩展性。对属性机制或对POSIX.1-2024卷中定义的任何属性对象的未来扩展必须谨慎进行，以免影响二进制兼容性。

属性对象，即使通过 `malloc()` 等动态分配函数分配，也可能在编译时固定其大小。这意味着，例如，在对 `pthread_attr_t` 有扩展的实现中，`pthread_create()` 不能查看二进制应用程序假设有效的区域之外。这建议实现应该在属性对象中维护大小字段，以及可能的版本信息，如果要容纳不同方向（可能由不同供应商）的扩展。

如果实现检测到 `pthread_attr_destroy()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，建议函数应该失败并报告 `[EINVAL]` 错误。

如果实现检测到 `pthread_attr_init()` 的 `attr` 参数指定的值引用已初始化的线程属性对象，建议函数应该失败并报告 `[EBUSY]` 错误。

未来方向

无。

另请参阅

`pthread_create()`、`malloc()`

POSIX.1-2024的Shell和实用程序卷，关于XSH系统接口选项的第2.9.2节

版权

本文的部分内容根据IEEE Std 1003.1-2024，信息技术标准——便携式操作系统接口（POSIX），The Open Group基础规范问题7，重新打印并电子复制，版权所有（C）2024，由电气和电子工程师协会Inc和The Open Group拥有。如果此版本与原始IEEE和The Open Group标准之间存在任何差异，应以原始IEEE和The Open Group 标准为准。原始标准可以在线获取：<https://pubs.opengroup.org/onlinepubs/9799919799/>。

信息性文本结束。

1.107. `pthread_attr_getdetachstate`, `pthread_attr_setdetachstate` — 获取和设置分离状态属性

概要

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

描述

`detachstate` 属性控制线程是否以分离状态创建。如果线程以分离状态创建，那么使用新创建线程的 ID 调用 `pthread_detach()` 或 `pthread_join()` 函数是错误的。

`pthread_attr_getdetachstate()` 和 `pthread_attr_setdetachstate()` 函数分别用于获取和设置 `attr` 对象中的 `detachstate` 属性。

对于 `pthread_attr_getdetachstate()`，`detachstate` 应被设置为 `PTHREAD_CREATE_DETACHED` 或 `PTHREAD_CREATE_JOINABLE`。

对于 `pthread_attr_setdetachstate()`，应用程序应将 `detachstate` 设置为 `PTHREAD_CREATE_DETACHED` 或 `PTHREAD_CREATE_JOINABLE`。

值为 `PTHREAD_CREATE_DETACHED` 将导致所有使用 `attr` 创建的线程处于分离状态，而使用 `PTHREAD_CREATE_JOINABLE` 值将导致所有使用 `attr` 创建的线程处于可连接状态。`detachstate` 属性的默认值应为 `PTHREAD_CREATE_JOINABLE`。

如果传递给 `pthread_attr_getdetachstate()` 或 `pthread_attr_setdetachstate()` 的 `attr` 参数值不指向已初始化的线程属性对象，则行为未定义。

返回值

成功完成后，`pthread_attr_getdetachstate()` 和 `pthread_attr_setdetachstate()` 应返回 0 值；否则，应返回错误编号以指示错误。

如果成功，`pthread_attr_getdetachstate()` 函数会将 `detachstate` 属性的值存储在 `detachstate` 中。

错误

`pthread_attr_setdetachstate()` 函数在以下情况会失败：

- `[EINVAL]`
- `detachstate` 的值无效

这些函数不应返回 `[EINTR]` 错误代码。

以下章节为参考信息。

示例

获取分离状态属性

本示例展示如何获取线程属性对象的 `detachstate` 属性。

```
#include <pthread.h>

pthread_attr_t thread_attr;
int          detachstate;
int          rc;

/* 初始化 thread_attr 的代码 */
...
rc = pthread_attr_getdetachstate(&thread_attr, &detachstate);
if (rc != 0) {
    /* 处理错误 */
    ...
}
else {
```

```
/* detachstate 的合法值：  
 * PTHREAD_CREATE_DETACHED 或 PTHREAD_CREATE_JOINABLE  
 */  
...  
}
```

应用程序使用

无。

基本原理

如果实现检测到传递给 `pthread_attr_getdetachstate()` 或 `pthread_attr_setdetachstate()` 的 `attr` 参数值不指向已初始化的线程属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

未来方向

无。

另请参阅

- `pthread_attr_destroy()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- `<pthread.h>`

变更历史

首次在 Issue 5 中发布。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数被标记为线程选项的一部分。

规范文本被更新以避免对应用程序要求使用"必须"一词。

应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/72, 向示例章节添加了示例。

应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/73, 更新错误章节以包含可选的 **[EINVAL]** 错误。

Issue 7

`pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数从线程选项移动到基本规范。

针对未初始化线程属性对象的 **[EINVAL]** 错误被移除；此条件会导致未定义行为。

1.108. `pthread_attr_getguardsize`, `pthread_attr_setguardsize` — 获取和设置线程保护大 小属性

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                               size_t *restrict guardsize);

int pthread_attr_setguardsize(pthread_attr_t *attr,
                               size_t guardsize);
```

DESCRIPTION

`pthread_attr_getguardsize()` 函数应获取 `attr` 对象中的 `guardsize` 属性。该属性应通过 `guardsize` 参数返回。

`pthread_attr_setguardsize()` 函数应设置 `attr` 对象中的 `guardsize` 属性。此属性的新值应从 `guardsize` 参数获取。如果 `guardsize` 为零，则不应为使用 `attr` 创建的线程提供保护区。如果 `guardsize` 大于零，则应为每个使用 `attr` 创建的线程提供至少大小为 `guardsize` 字节的保护区。

`guardsize` 属性控制创建线程的栈的保护区域大小。`guardsize` 属性提供对栈指针溢出的保护。如果线程的栈是使用保护机制创建的，实现会在栈的溢出端分配额外的内存，作为栈指针溢出的缓冲区。如果应用程序溢出到这个缓冲区中，将会产生错误（可能会向线程传递 `SIGSEGV` 信号）。

一致实现可以将 `guardsize` 中的值向上舍入为可配置系统变量 `{PAGESIZE}` 的倍数（参见 `<sys/mman.h>`）。如果实现将 `guardsize` 的值向上舍入为 `{PAGESIZE}` 的倍数，那么指定 `attr` 的 `pthread_attr_getguardsize()` 调用应在前一个 `pthread_attr_setguardsize()` 函数调用指定的保护大小存储到 `guardsize` 参数中。

`guardsize` 属性的默认值由实现定义。

如果设置了 `stackaddr` 属性（即调用者正在分配和管理自己的线程栈），则应忽略 `guardsize` 属性，实现不应提供保护。在这种情况下，应用程序有责任

管理栈溢出以及栈的分配和管理。

如果 `pthread_attr_getguardsize()` 或 `pthread_attr_setguardsize()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_attr_getguardsize()` 和 `pthread_attr_setguardsize()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

这些函数可能在以下情况下失败：

- `EINVAL`
- 参数 `guardsize` 无效。

这些函数不应返回 `[EINTR]` 错误代码。

EXAMPLES

获取 `guardsize` 属性

此示例显示如何获取线程属性对象的 `guardsize` 属性。

```
#include <pthread.h>

pthread_attr_t thread_attr;
size_t guardsize;
int rc;

/* 初始化 thread_attr 的代码 */
...
rc = pthread_attr_getguardsize(&thread_attr, &guardsize);
if (rc != 0) {
    /* 处理错误 */
    ...
}
else {
    if (guardsize > 0) {
```

```
    /* 提供了至少 guardsize 字节大小的保护区 */
    ...
}
else {
    /* 没有提供保护区 */
    ...
}
}
```

APPLICATION USAGE

无。

RATIONALE

向应用程序提供 `guardsize` 属性有两个原因：

1. 溢出保护可能导致系统资源浪费。创建大量线程且知道其线程永远不会溢出栈的应用程序可以通过关闭保护区来节省系统资源。
2. 当线程在栈上分配大型数据结构时，可能需要大的保护区来检测栈溢出。

保护区的默认大小由实现定义，因为在支持非常大页面大小的系统上，如果默认要求至少一个保护页面，开销可能会很大。

如果实现检测到 `pthread_attr_getguardsize()` 或 `pthread_attr_setguardsize()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `<pthread.h>`
- `<sys/mman.h>`

CHANGE HISTORY

首次发布于 Issue 5。

Issue 6

- 在 ERRORS 部分中，删除了第三个 `[EINVAL]` 错误条件，因为它已被第二个错误条件涵盖。
- 为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getguardsize()` 原型添加了 `restrict` 关键字。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/74，更新 ERRORS 部分以删除 `[EINVAL]` 错误 ("属性 `attr` 无效。")，并用可选的 `[EINVAL]` 错误替换它。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/76，向 EXAMPLES 部分添加了示例。

Issue 7

- 应用了 SD5-XSH-ERN-111，删除了 DESCRIPTION 中对 `stack` 属性的引用。
 - 应用了 SD5-XSH-ERN-175，更新 DESCRIPTION 以注意保护区的默认大小由实现定义。
 - `pthread_attr_getguardsize()` 和 `pthread_attr_setguardsize()` 函数从 XSI 选项移动到 Base。
 - 删除了未初始化线程属性对象的 `[EINVAL]` 错误；此条件导致未定义行为。
-

1.109. `pthread_attr_getinheritsched`, `pthread_attr_setinheritsched` — 获取和设置 inheritsched 属性 (实时线程)

概要

```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict
                                  int *restrict inheritsched);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                  int inheritsched);
```

描述

`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数分别用于获取和设置 `attr` 参数中的 `inheritsched` 属性。

当属性对象被 `pthread_create()` 使用时, `inheritsched` 属性决定了所创建线程的其他调度属性应如何设置。

`inheritsched` 支持的值为:

- `PTHREAD_INHERIT_SCHED`
- 指定线程调度属性应从创建线程继承, 此 `attr` 参数中的调度属性将被忽略。
- `PTHREAD_EXPLICIT_SCHED`
- 指定线程调度属性应设置为此属性对象中的相应值。

符号 `PTHREAD_INHERIT_SCHED` 和 `PTHREAD_EXPLICIT_SCHED` 在 `<pthread.h>` 头文件中定义。

POSIX.1-2024 定义的以下线程调度属性受 `inheritsched` 属性影响: 调度策略 (`schedpolicy`)、调度参数 (`schedparam`) 和调度竞争范围 (`contentionscope`)。

如果传递给 `pthread_attr_getinheritsched()` 或 `pthread_attr_setinheritsched()` 的 `attr` 参数值不指向已初始化的线程属性对象，则行为未定义。

返回值

如果成功，`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_attr_setinheritsched()` 函数在以下情况下应失败：

- `[ENOTSUP]`
- 尝试将属性设置为不支持的值。

`pthread_attr_setinheritsched()` 函数在以下情况下可能失败：

- `[EINVAL]`
- `inheritsched` 的值无效。

这些函数不应返回 `[EINTR]` 错误码。

示例

无。

应用用法

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前运行的线程。

有关线程调度属性及其默认设置的更多详细信息，请参阅 [2.9.4 线程调度](#)。

原理

如果实现检测到传递给 `pthread_attr_getinheritsched()` 或 `pthread_attr_setinheritsched()` 的 `attr` 参数值不指向已初始化的线程

属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

未来方向

无。

参见

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`
- XBD `<pthread.h>`
- XBD `<sched.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展保持一致而包含在内。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数被标记为线程和线程执行调度选项的一部分。

由于如果实现不支持线程执行调度选项，则不需要提供存根，因此移除了 `[ENOSYS]` 错误条件。

为与 ISO/IEC 9899:1999 标准保持一致，向 `pthread_attr_getinheritsched()` 原型添加了 `restrict` 关键字。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/75，澄清了描述中 `inheritsched` 的值，并在错误部分添加了两个可选的 `[EINVAL]` 错误，用于检查 `attr` 是否指向未初始化的线程属性对象。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/77，在应用用法部分添加了对 [2.9.4 线程调度](#) 的引用。

Issue 7

`pthread_attr_getinheritsched()` 和
`pthread_attr_setinheritsched()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。

移除了未初始化线程属性对象的 `[EINVAL]` 错误；此条件导致未定义行为。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0450 [314]。

应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0262 [757]。

1.110. pthread_attr_getschedparam, pthread_attr_setschedparam

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                               struct sched_param *restrict param);

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                               const struct sched_param *restrict param);
```

DESCRIPTION

`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数分别用于获取和设置 `attr` 参数中的调度参数属性。`param` 结构体内容在 `<sched.h>` 头文件中定义。对于 SCHED_FIFO 和 SCHED_RR 策略，`param` 中唯一的必需成员是 `sched_priority`。

[TSP] 对于 SCHED_SPORADIC 策略，`param` 结构体的必需成员包括 `sched_priority` 、 `sched_ss_low_priority` 、 `sched_ss_repl_period` 、 `sched_ss_init_budget` 和 `sched_ss_max_repl`。指定的 `sched_ss_repl_period` 需要大于或等于指定的 `sched_ss_init_budget` 函数才能成功；如果不是，则函数将失败。`sched_ss_max_repl` 的值必须在包含范围 [1,{SS_REPL_MAX}] 内函数才能成功；否则，函数将失败。`sched_ss_repl_period` 和 `sched_ss_init_budget` 值是按此函数提供的方式存储还是被四舍五入以与所用时钟的分辨率对齐，这是未指定的。

如果 `pthread_attr_getschedparam()` 或 `pthread_attr_setschedparam()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数应返回零；否则，应返回错误码以指示错误。

ERRORS

`pthread_attr_setschedparam()` 函数在以下情况下应失败：

- `ENOTSUP`
- 尝试将属性设置为不支持的值。

`pthread_attr_setschedparam()` 函数在以下情况下可能失败：

- `EINVAL`
- `param` 的值无效。

这些函数不应返回错误码 [EINTR]。

以下章节为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前运行的线程。

RATIONALE

如果实现检测到 `pthread_attr_getschedparam()` 或 `pthread_attr_setschedparam()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_create()`

XBD `<pthread.h>`, `<sched.h>`

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含在内。

Issue 6

`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数被标记为线程选项的一部分。

为与 IEEE Std 1003.1d-1999 对齐而添加了 SCHED_SPORADIC 调度策略。

为与 ISO/IEC 9899:1999 标准对齐，在 `pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 原型中添加了 `restrict` 关键字。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/78，更新 ERRORS 章节以包含当 `attr` 引用未初始化线程属性对象时的可选错误。

Issue 7

`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数从线程选项移动到基础部分。

应用 Austin Group Interpretation 1003.1-2001 #119，阐明了对 `sched_ss_repl_period` 和 `sched_ss_init_budget` 值的准确性要求。

移除了针对未初始化线程属性对象的 [EINVAL] 错误；这种情况会导致未定义行为。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0451 [314]。

信息性文本结束。

1.111. `pthread_attr_getschedpolicy`, `pthread_attr_setschedpolicy`

SYNOPSIS (概要)

```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict
                                int *restrict policy);

int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
```

DESCRIPTION (描述)

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数分别用于获取和设置 attr 参数中的调度策略属性。

支持的 policy 值应包括 SCHED_FIFO、SCHED_RR 和 SCHED_OTHER，这些值在 `<sched.h>` 头文件中定义。当以调度策略 SCHED_FIFO、SCHED_RR 或 SCHED_SPORADIC 执行的线程在互斥锁上等待时，它们应在互斥锁解锁时按优先级顺序获取互斥锁。

如果传递给 `pthread_attr_getschedpolicy()` 或 `pthread_attr_setschedpolicy()` 的 attr 参数值不引用已初始化的线程属性对象，则行为未定义。

RETURN VALUE (返回值)

如果成功，`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数应返回零；否则，应返回错误码以指示错误。

ERRORS (错误)

`pthread_attr_setschedpolicy()` 函数在以下情况下可能失败：

- `ENOTSUP`

- 尝试将属性设置为不支持的值。

`pthread_attr_setschedpolicy()` 函数在以下情况下可能失败：

- `EINVAL`
- `policy` 的值无效。

这些函数不应返回 `[EINTR]` 错误码。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前正在运行的线程。

有关线程调度属性及其默认设置的更多详细信息，请参阅 [2.9.4 线程调度](#)。

RATIONALE (基本原理)

如果实现检测到传递给 `pthread_attr_getschedpolicy()` 或 `pthread_attr_setschedpolicy()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedparam()`
- `pthread_create()`

CHANGE HISTORY (变更历史)

首次在 Issue 5 中发布。为了与 POSIX 线程扩展对齐而包含。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数被标记为线程和线程执行调度选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持线程执行调度选项，则不需要提供存根。

为了与 IEEE Std 1003.1d-1999 对齐，添加了 SCHED_SPORADIC 调度策略。

为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getschedpolicy()` 原型添加了 **restrict** 关键字。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/79，在 APPLICATION USAGE 部分添加了对 [2.9.4 线程调度](#) 的引用。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/80，更新 ERRORS 部分以包含当 attr 引用未初始化线程属性对象时的可选错误。

Issue 7

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在已成为基础的一部分。

对于未初始化线程属性对象的 [EINVAL] 错误已被移除；此条件导致未定义行为。

应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0452 [314]。

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0263 [757]。

1.112. `pthread_attr_getscope`, `pthread_attr_setscope`

概要

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                           int *restrict contentionscope);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

描述

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数分别用于获取和设置 *attr* 对象中的 *contentionscope* 属性。

contentionscope 属性可以取值为 `PTHREAD_SCOPE_SYSTEM`，表示系统调度竞争范围；或者 `PTHREAD_SCOPE_PROCESS`，表示进程调度竞争范围。符号 `PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS` 在 `<pthread.h>` 头文件中定义。

如果传递给 `pthread_attr_getscope()` 或 `pthread_attr_setscope()` 的 *attr* 参数值不引用已初始化的线程属性对象，则行为未定义。

返回值

如果成功，`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数应返回零；否则，应返回错误码以指示错误。

错误

`pthread_attr_setscope()` 函数在以下情况下应失败：

- **[ENOTSUP]**

尝试将属性设置为不支持的值。

`pthread_attr_setscope()` 函数在以下情况下可能失败：

- **[EINVAL]**

contentionscope 的值无效。

这些函数不应返回 **[EINTR]** 错误码。

以下各节为提供参考信息。

示例

无。

应用用法

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前正在运行的线程。

关于线程调度属性及其默认设置的更多详细信息，请参阅 [2.9.4 线程调度](#)。

基本原理

如果实现检测到传递给 `pthread_attr_getscope()` 或 `pthread_attr_setscope()` 的 *attr* 参数值不引用已初始化的线程属性对象，建议函数应失败并报告 **[EINVAL]** 错误。

未来方向

无。

另请参阅

- `pthread_attr_destroy()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`

变更历史

首次发布于 Issue 5。为了与 POSIX 线程扩展对齐而包含在内。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数被标记为线程和线程执行调度选项的一部分。

如果实现不支持线程执行调度选项，则不需要提供存根，因此删除了 **[ENOSYS]** 错误条件。

为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getscope()` 原型添加了 `restrict` 关键字。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/81，在应用用法部分添加了对 [2.9.4 线程调度](#) 的引用。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/82，更新错误部分以包含当 *attr* 引用未初始化线程属性对象时的可选错误。

Issue 7

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。

删除了未初始化线程属性对象的 **[EINVAL]** 错误；此条件导致未定义行为。

应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0453 [314]。

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0264 [757]。

1.113. `pthread_attr_getstack`, `pthread_attr_setstack`

SYNOPSIS

```
[TSA TSS]
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stackaddr,
                         size_t stacksize);
```

DESCRIPTION

`pthread_attr_getstack()` 和 `pthread_attr_setstack()` 函数分别用于获取和设置 `attr` 对象中的线程创建栈属性 `stackaddr` 和 `stacksize`。

栈属性指定要用于创建线程栈的存储区域。存储的基址（最低可寻址字节）应为 `stackaddr`，存储的大小应为 `stacksize` 字节。`stacksize` 应至少为 `{PTHREAD_STACK_MIN}`。如果 `stackaddr` 不满足实现定义的对齐要求，`pthread_attr_setstack()` 函数可能失败并返回 `[EINVAL]` 错误。由 `stackaddr` 和 `stacksize` 描述的栈内的所有页应都可被线程读写。

如果在设置 `stackaddr` 属性之前调用 `pthread_attr_getstack()` 函数，其行为是未指定的。

如果传递给 `pthread_attr_getstack()` 或 `pthread_attr_setstack()` 的 `attr` 参数值不指向已初始化的线程属性对象，则行为是未定义的。

RETURN VALUE

成功完成后，这些函数应返回 0 值；否则，应返回错误号以指示错误。

如果成功，`pthread_attr_getstack()` 函数应将栈属性值存储在 `stackaddr` 和 `stacksize` 中。

ERRORS

`pthread_attr_setstack()` 函数在以下情况应失败：

- **[EINVAL]**
- `stacksize` 的值小于 {PTHREAD_STACK_MIN} 或超过实现定义的限制。

`pthread_attr_setstack()` 函数在以下情况可能失败：

- **[EINVAL]**
- `stackaddr` 的值没有适当的对齐以用作栈，或 `((char *)stackaddr + stacksize)` 缺乏适当的对齐。
- **[EACCES]**
- 由 `stackaddr` 和 `stacksize` 描述的栈页不可被线程同时读写。

这些函数不应返回 [EINTR] 错误码。

EXAMPLES

无。

APPLICATION USAGE

这些函数适用于应用程序必须将线程的栈放置在内存的特定区域的环境。

虽然看起来应用程序可以通过在指定栈区域外提供受保护页来检测栈溢出，但这无法可移植地实现。实现可以自由地将线程的初始栈指针放置在指定区域内的任何位置，以适应机器的栈指针行为和分配要求。此外，在某些体系结构上，如 IA-64，“溢出”可能意味着在区域内分配的两个独立栈指针将在区域的中间某处重叠。

在成功调用 `pthread_attr_setstack()` 后，由 `stackaddr` 参数指定的存储区域由实现控制，如 2.9.8 Use of Application-Managed Thread Stacks 中所述。

`stackaddr` 属性的规范存在一些歧义，使得这些函数的可移植使用变得不可能。例如，标准允许实现对 `stackaddr` 强加任意的对齐要求。应用程序不能假设从 `malloc()` 获得的缓冲区是适当对齐的。注意，虽然传递给 `pthread_attr_setstack()` 的 `stacksize` 值必须满足对齐要求，但对于

`pthread_attr_setstacksize()` 则不同，如果需要实现必须增加指定的大小以实现适当的对齐。

RATIONALE

如果实现检测到传递给 `pthread_attr_getstack()` 或 `pthread_attr_setstack()` 的 `attr` 参数值不指向已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 6。作为 XSI 选项的一部分开发，并通过 IEEE PASC Interpretation 1003.1 #101 纳入 BASE。

应用 IEEE Std 1003.1-2001/Cor 2-2004 中的项目 XSH/TC2/D6/83，更新 APPLICATION USAGE 部分以引用 [2.9.8 Use of Application-Managed Thread Stacks](#)。

应用 IEEE Std 1003.1-2001/Cor 2-2004 中的项目 XSH/TC/D6/84，更新 ERRORS 部分以包含当 `attr` 指向未初始化线程属性对象时的可选错误。

Issue 7

应用 SD5-XSH-ERN-66，修正 [EINVAL] 错误条件中对 `attr` 的使用。

应用 Austin Group Interpretation 1003.1-2001 #057，阐明在设置 `stackaddr` 属性之前调用函数时的行为。

应用 SD5-XSH-ERN-157，更新 APPLICATION USAGE 部分。

在 DESCRIPTION 和 APPLICATION USAGE 部分更新了 `stackaddr` 属性的描述。

删除了针对未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。

信息性文本结束。

1.114. `pthread_attr_getstackaddr`, `pthread_attr_setstackaddr`

概要 (SYNOPSIS)

```
[0B] #include <pthread.h>

int pthread_attr_getstackaddr(const pthread_attr_t *restrict attr,
                               void **restrict stackaddr);

int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stackaddr);
```

描述 (DESCRIPTION)

`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 函数分别用于获取和设置 `attr` 对象中的线程创建 `stackaddr` 属性。

`stackaddr` 属性指定用于所创建线程栈的存储位置。该存储的大小应至少为 {PTHREAD_STACK_MIN}。

返回值 (RETURN VALUE)

成功完成后，`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 应返回值 0；否则，应返回错误码以指示错误。

如果成功，`pthread_attr_getstackaddr()` 函数将 `stackaddr` 属性值存储在 `stackaddr` 中。

错误 (ERRORS)

这些函数可能会在以下情况下失败：

- [EINVAL]

`attr` 指定的值不引用已初始化的线程属性对象。

这些函数不应返回 [EINTR] 错误码。

应用程序使用 (APPLICATION USAGE)

`stackaddr` 属性的规范存在一些歧义，使得这些接口的可移植使用变得不可能。将单个地址参数描述为"栈"并未指定地址与该地址所暗示的"栈"之间的特定关系。例如，该地址可以被视为用作栈的缓冲区的低内存地址，或者可以被视为用作新线程的初始栈指针寄存器值的地址。这两者并不相同，除非是在栈从低内存向高内存"向上"增长的机器上，并且在该机器上"推入"操作首先将值存储在内存中，然后递增栈指针寄存器。此外，在栈从高内存向低内存"向下"增长的机器上，将地址解释为"低内存"地址需要确定栈的预期大小。IEEE Std 1003.1-2001 引入了新的接口 `pthread_attr_setstack()` 和 `pthread_attr_getstack()` 来解决这些歧义。

在成功调用 `pthread_attr_setstackaddr()` 后，`stackaddr` 参数指定的存储区域由实现控制，如应用程序管理的线程栈的使用中所述。

另请参见 (SEE ALSO)

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstack()`
- `pthread_attr_getstacksize()`
- `pthread_attr_setstack()`
- `pthread_create()`
- `<limits.h>`
- `<pthread.h>`

更改历史 (CHANGE HISTORY)

首次在 Issue 5 中发布。包含在内以与 POSIX 线程扩展对齐。

Issue 6

`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 函数被标记为线程和线程栈地址属性选项的一部分。

为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getstackaddr()` 原型添加了 `restrict` 关键字。

这些函数被标记为过时。

应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/85, 更新了应用程序使用部分以引用应用程序管理的线程栈的使用。

应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/86, 更新了错误部分以包含当 **attr** 引用未初始化的线程属性对象时的可选错误。

1.115. `pthread_attr_getstacksize`, `pthread_attr_setstacksize` — 获取和设置栈大小属性

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                               size_t *restrict stacksize);

int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

DESCRIPTION

`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数分别用于获取和设置 *attr* 对象中的线程创建 *stacksize* (栈大小) 属性。

stacksize 属性定义了为所创建线程栈分配的最小栈大小 (以字节为单位)。

如果传递给 `pthread_attr_getstacksize()` 或 `pthread_attr_setstacksize()` 的 *attr* 参数值不指向已初始化的线程属性对象，则行为是未定义的。

RETURN VALUE

成功完成后，`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 应返回值 0；否则，应返回错误码以指示错误。

如果成功，`pthread_attr_getstacksize()` 函数会将 *stacksize* 属性值存储在 *stacksize* 中。

ERRORS

`pthread_attr_setstacksize()` 函数在以下情况下应失败：

- **[EINVAL]**

stacksize 的值小于 {PTHREAD_STACK_MIN} 或超过系统强制的限制。

这些函数不应返回 [EINTR] 错误码。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

如果实现检测到传递给 `pthread_attr_getstacksize()` 或 `pthread_attr_setstacksize()` 的 *attr* 参数值不指向已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数被标记为线程和线程栈大小属性选项的一部分。
- 为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getstacksize()` 原型添加了 **restrict** 关键字。
- 应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/43，将 SYNOPSIS 中的边距代码从 TSA 更正为 TSS，并将 CHANGE HISTORY 从“线程栈地址属性”选项更新为“线程栈大小属性”选项。
- 应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/87，更新 ERRORS 部分以包含当 *attr* 指向未初始化线程属性对象时的可选错误。

Issue 7

- `pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数仅被标记为线程栈大小属性选项的一部分，因为线程选项现在是基础的一部分。
 - 移除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。
 - 应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0265 [757]。
-

1.116. `pthread_attr_init`, `pthread_attr_destroy` - 初始化和销毁线程属性对象

概要

```
#include <pthread.h>

int pthread_attr_init(pthread_attr_t *attr);
int pthread_attr_destroy(pthread_attr_t *attr);
```

描述

`pthread_attr_destroy()` 函数应销毁一个线程属性对象。实现可能导致 `pthread_attr_destroy()` 将 `attr` 设置为实现定义的无效值。已销毁的 `attr` 属性对象可以使用 `pthread_attr_init()` 重新初始化；在对象被销毁后以其他方式引用该对象的结果是未定义的。

`pthread_attr_init()` 函数应使用给定实现所使用的所有单个属性的默认值来初始化线程属性对象 `attr`。

当由 `pthread_create()` 使用时，生成的属性对象（可能通过设置单个属性值来修改）定义了所创建线程的属性。单个属性对象可以在多次并发调用 `pthread_create()` 中使用。如果调用 `pthread_attr_init()` 时指定了已初始化的 `attr` 属性对象，则结果是未定义的。

如果传递给 `pthread_attr_destroy()` 的 `attr` 参数值不引用已初始化的线程属性对象，则行为是未定义的。

返回值

成功完成后，`pthread_attr_init()` 和 `pthread_attr_destroy()` 应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_attr_init()` 和 `pthread_attr_destroy()` 函数在以下情况下应失败：

- **ENOMEM** - 存在的内存不足以初始化线程属性对象。

`pthread_attr_init()` 函数在以下情况下可能失败：

- **EBUSY** - `attr` 指定的值引用已初始化的线程属性对象。

`pthread_attr_destroy()` 函数在以下情况下可能失败：

- **EINVAL** - `attr` 指定的值不引用已初始化的线程属性对象。

这些函数不应返回 `EINTR` 错误码。

示例

无。

应用程序用法

为线程、互斥量和条件变量提供了属性对象，作为一种机制，用于支持这些领域中可能的未来标准化，而无需更改函数本身。

属性对象提供了线程可配置方面的清晰隔离。例如，"栈大小"是线程的一个重要属性，但它无法以可移植的方式表达。在移植线程程序时，通常需要调整栈大小。使用属性对象可以通过允许将更改隔离在单个位置来提供帮助，而不是分散在每次线程创建的实例中。

属性对象可用于设置具有相似属性的线程"类"；例如，"具有大栈和高优先级的线程"或"具有最小栈的线程"。这些类可以在单个位置定义，然后在需要创建线程的任何地方引用。对"类"决策的更改变得直接明了，并且不需要对每个 `pthread_create()` 调用进行详细分析。

基本原理

属性对象被定义为不透明类型，以有助于扩展性。如果这些对象被指定为结构，则在属性对象被扩展时，添加新属性将强制重新编译所有多线程程序；如果不同的程序组件由不同的供应商提供，这可能是不可能的。

此外，不透明的属性对象提供了提高性能的机会。属性的有效性可以在设置属性时检查一次，而不是在每次创建线程时都检查。实现通常需要缓存创建成本高昂的内核对象。不透明属性对象提供了一种高效的机制来检测缓存的对象何时因属性更改而失效。

由于赋值在给定的不透明类型上不一定被定义，实现定义的默认值无法以可移植的方式定义。这个问题的解决方案是允许属性对象通过属性对象初始化函数

动态初始化，以便实现可以自动提供默认值。

栈大小被定义为可选属性，因为栈的概念本身就是机器依赖的。例如，一些实现可能无法更改栈的大小，而其他实现可能不需要，因为栈页可能是不连续的，可以按需分配和释放。

属性机制在很大程度上是为可扩展性而设计的。对属性机制或 POSIX.1-2024 卷中定义的任何属性对象的未来扩展必须谨慎进行，以免影响二进制兼容性。

属性对象，即使是通过诸如 `malloc()` 等动态分配函数分配的，其大小也可能在编译时固定。这意味着，例如，在具有 `pthread_attr_t` 扩展的实现中的 `pthread_create()` 不能查看超出二进制应用程序假设为有效的区域。这表明实现应在属性对象中维护大小字段，以及可能的版本信息，如果要适应不同方向（可能由不同供应商）的扩展。

如果实现检测到传递给 `pthread_attr_destroy()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

如果实现检测到传递给 `pthread_attr_init()` 的 `attr` 参数值引用已初始化的线程属性对象，建议函数应失败并报告 `[EBUSY]` 错误。

未来方向

无。

另请参阅

`pthread_create()`, `malloc()`

POSIX.1-2024 基础定义卷, `<pthread.h>`

更改历史

首次发布于 Issue 5。包含在 Issue 6 (IEEE Std 1003.1, 2004) 中。

1.117. `pthread_attr_getdetachstate`, `pthread_attr_setdetachstate` — 获取和设置分离状态属性

概要

```
#include <pthread.h>

int pthread_attr_getdetachstate(const pthread_attr_t *attr, int *detachstate);
int pthread_attr_setdetachstate(pthread_attr_t *attr, int detachstate);
```

描述

`detachstate` 属性控制线程是否以分离状态创建。如果线程以分离状态创建，那么使用 `pthread_detach()` 或 `pthread_join()` 函数操作新创建线程的 ID 是一个错误。

`pthread_attr_getdetachstate()` 和 `pthread_attr_setdetachstate()` 函数分别用于获取和设置 `attr` 对象中的 `detachstate` 属性。

对于 `pthread_attr_getdetachstate()`，`detachstate` 应被设置为 `PTHREAD_CREATE_DETACHED` 或 `PTHREAD_CREATE_JOINABLE`。

对于 `pthread_attr_setdetachstate()`，应用程序应将 `detachstate` 设置为 `PTHREAD_CREATE_DETACHED` 或 `PTHREAD_CREATE_JOINABLE`。

值为 `PTHREAD_CREATE_DETACHED` 将导致所有使用 `attr` 创建的线程处于分离状态，而使用值为 `PTHREAD_CREATE_JOINABLE` 将导致所有使用 `attr` 创建的线程处于可连接状态。`detachstate` 属性的默认值为 `PTHREAD_CREATE_JOINABLE`。

如果传递给 `pthread_attr_getdetachstate()` 或 `pthread_attr_setdetachstate()` 的 `attr` 参数值不引用已初始化的线程属性对象，则行为未定义。

返回值

成功完成时，`pthread_attr_getdetachstate()` 和 `pthread_attr_setdetachstate()` 应返回值 0；否则，应返回错误编号以指示错误。

如果成功，`pthread_attr_getdetachstate()` 函数将 `detachstate` 属性的值存储在 `detachstate` 中。

错误

`pthread_attr_setdetachstate()` 函数应在以下情况失败：

- `[EINVAL]`
- `detachstate` 的值无效

这些函数不应返回 `[EINTR]` 错误代码。

示例

获取分离状态属性

本示例演示如何获取线程属性对象的 `detachstate` 属性。

```
#include <pthread.h>

pthread_attr_t thread_attr;
int          detachstate;
int          rc;

/* 初始化 thread_attr 的代码 */
...
rc = pthread_attr_getdetachstate(&thread_attr, &detachstate);
if (rc != 0) {
    /* 处理错误 */
    ...
} else {
    /* detachstate 的合法值为：
     * PTHREAD_CREATE_DETACHED 或 PTHREAD_CREATE_JOINABLE
     */
    ...
}
```

应用程序使用

无。

基本原理

如果实现检测到传递给 `pthread_attr_getdetachstate()` 或 `pthread_attr_setdetachstate()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

另请参阅

- `pthread_attr_destroy()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- `<pthread.h>`

变更历史

首次发布于 Issue 5

为与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数被标记为线程选项的一部分。
- 规范性文本已更新，避免在应用程序要求中使用"必须"一词。
- 应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/72，在示例部分添加示例。

- 应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/73, 更新错误部分以包含可选的 [EINVAL] 错误。

Issue 7

- `pthread_attr_setdetachstate()` 和 `pthread_attr_getdetachstate()` 函数从线程选项移动到基础。
 - 移除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。
-

1.118. pthread_attr_getguardsize

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getguardsize(const pthread_attr_t *restrict attr,
                               size_t *restrict guardsize);
int pthread_attr_setguardsize(pthread_attr_t *attr,
                               size_t guardsize);
```

DESCRIPTION

`pthread_attr_getguardsize()` 函数应获取 `attr` 对象中的 `guardsize` 属性。此属性应在 `guardsize` 参数中返回。

`pthread_attr_setguardsize()` 函数应设置 `attr` 对象中的 `guardsize` 属性。此属性的新值应从 `guardsize` 参数获取。如果 `guardsize` 为零，则使用 `attr` 创建的线程不应提供警戒区。如果 `guardsize` 大于零，则使用 `attr` 创建的每个线程应提供至少大小为 `guardsize` 字节的警戒区。

`guardsize` 属性控制所创建线程栈的警戒区大小。`guardsize` 属性提供对栈指针溢出的保护。如果线程的栈是在警戒保护下创建的，实现会在栈的溢出端分配额外内存作为缓冲区，以防止栈指针的栈溢出。如果应用程序溢出到此缓冲区中，则应产生错误（可能导致向线程传递 SIGSEGV 信号）。

一致性实现可能将 `guardsize` 中包含的值向上舍入到可配置系统变量 {PAGESIZE} 的倍数（参见 `<sys/mman.h>`）。如果实现将 `guardsize` 的值向上舍入到 {PAGESIZE} 的倍数，则指定 `attr` 的 `pthread_attr_getguardsize()` 调用应在 `guardsize` 参数中存储由前一个 `pthread_attr_setguardsize()` 函数调用指定的警戒区大小。

`guardsize` 属性的默认值由实现定义。

如果已设置 `stackaddr` 属性（即调用者正在分配和管理自己的线程栈），则应忽略 `guardsize` 属性，实现不应提供保护。在这种情况下，应用程序负责管理栈溢出以及栈分配和管理。

如果 `pthread_attr_getguardsize()` 或 `pthread_attr_setguardsize()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_attr_getguardsize()` 和 `pthread_attr_setguardsize()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS

这些函数可能在以下情况下失败：

- **[EINVAL]** - 参数 `guardsize` 无效。

这些函数不应返回 [EINTR] 错误代码。

以下部分为提供信息的内容。

EXAMPLES

获取警戒区大小属性

此示例显示如何获取线程属性对象的 `guardsize` 属性。

```
#include <pthread.h>

pthread_attr_t thread_attr;
size_t guardsize;
int rc;

/* 初始化 thread_attr 的代码 */
...
rc = pthread_attr_getguardsize (&thread_attr, &guardsize);
if (rc != 0) {
    /* 处理错误 */
    ...
}
else {
    if (guardsize > 0) {
        /* 提供了至少 guardsize 字节的警戒区 */
        ...
    }
}
```

```
else {
/* 未提供警戒区 */
...
}
}
```

APPLICATION USAGE

无。

RATIONALE

向应用程序提供 `guardsize` 属性有两个原因：

1. 溢出保护可能导致系统资源浪费。创建大量线程且知道其线程从不溢出栈的应用程序可以通过关闭警戒区来节省系统资源。
2. 当线程在栈上分配大型数据结构时，可能需要大型警戒区来检测栈溢出。

警戒区的默认大小由实现定义，因为在支持非常大页面大小的系统上，如果默认要求至少一个警戒页，开销可能很大。

如果实现检测到 `pthread_attr_getguardsize()` 或 `pthread_attr_setguardsize()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

XBD `<pthread.h>` , `<sys/mman.h>`

CHANGE HISTORY

首次发布于 Issue 5。

Issue 6

在 ERRORS 部分中，删除了第三个 [EINVAL] 错误条件，因为它已被第二个错误条件覆盖。

`restrict` 关键字被添加到 `pthread_attr_getguardsize()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/74，更新 ERRORS 部分以删除 [EINVAL] 错误 ("属性 `attr` 无效。")，并用可选的 [EINVAL] 错误替换。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/76，向 EXAMPLES 部分添加了示例。

Issue 7

应用了 SD5-XSH-ERN-111，删除了 DESCRIPTION 中对 `stack` 属性的引用。

应用了 SD5-XSH-ERN-175，更新 DESCRIPTION 以注意警戒区的默认大小由实现定义。

`pthread_attr_getguardsize()` 和 `pthread_attr_setguardsize()` 函数从 XSI 选项移动到 Base。

删除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。

1.119. `pthread_attr_getinheritsched`, `pthread_attr_setinheritsched` — 获取和设置 inheritsched 属性 (实时线程)

概要

```
#include <pthread.h>

int pthread_attr_getinheritsched(const pthread_attr_t *restrict
                                  int *restrict inheritsched);

int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                  int inheritsched);
```

描述

`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数分别用于获取和设置 `attr` 参数中的 `inheritsched` 属性。

当属性对象被 `pthread_create()` 使用时, `inheritsched` 属性决定了所创建线程的其他调度属性应如何设置。

`inheritsched` 支持的值包括:

- **PTHREAD_INHERIT_SCHED**

指定线程调度属性应继承自创建线程, 而此 `attr` 参数中的调度属性应被忽略。

- **PTHREAD_EXPLICIT_SCHED**

指定线程调度属性应设置为此属性对象中的相应值。

符号 `PTHREAD_INHERIT_SCHED` 和 `PTHREAD_EXPLICIT_SCHED` 在 `<pthread.h>` 头文件中定义。

由 POSIX.1-2024 定义的以下线程调度属性受 `inheritsched` 属性影响: 调度策略 (`schedpolicy`)、调度参数 (`schedparam`) 和调度竞争范围 (`contentionscope`)。

如果传递给 `pthread_attr_getinheritsched()` 或 `pthread_attr_setinheritsched()` 的 `attr` 参数值不引用已初始化的线程

属性对象，则行为未定义。

返回值

如果成功，`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数应返回零；否则，应返回错误号以指示错误。

错误

`pthread_attr_setinheritsched()` 函数在以下情况可能失败：

- **[ENOTSUP]**

试图将属性设置为不支持的值。

`pthread_attr_setinheritsched()` 函数在以下情况可能失败：

- **[EINVAL]**

`inheritsched` 的值无效。

这些函数不应返回 **[EINTR]** 错误码。

示例

无。

应用程序使用

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前运行的线程。

有关线程调度属性及其默认设置的更多详细信息，请参见 [2.9.4 线程调度](#)。

原理

如果实现检测到传递给 `pthread_attr_getinheritsched()` 或 `pthread_attr_setinheritsched()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应失败并报告 **[EINVAL]** 错误。

未来方向

无。

另请参见

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`
- `<pthread.h>`
- `<sched.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数被标记为线程和线程执行调度选项的一部分。

`[ENOSYS]` 错误条件已被移除，因为如果实现不支持线程执行调度选项，则无需提供存根。

`restrict` 关键字被添加到 `pthread_attr_getinheritsched()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/75，阐明了描述中 `inheritsched` 的值，并向错误部分添加了两个可选的 `[EINVAL]` 错误，用于检查 `attr` 是否引用未初始化的线程属性对象。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/77，在应用程序使用部分添加了对 2.9.4 线程调度 的引用。

Issue 7

`pthread_attr_getinheritsched()` 和 `pthread_attr_setinheritsched()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。

针对未初始化线程属性对象的 `[EINVAL]` 错误已被移除；这种情况会导致未定义行为。

应用了 POSIX.1-2008，技术更正 1，XSH/TC1-2008/0450 [314]。

应用了 POSIX.1-2008，技术更正 2，XSH/TC2-2008/0262 [757]。

1.120. `pthread_attr_getschedparam`, `pthread_attr_setschedparam` — 获取和设置调度参数属性

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedparam(const pthread_attr_t *restrict attr,
                                struct sched_param *restrict param);

int pthread_attr_setschedparam(pthread_attr_t *restrict attr,
                                const struct sched_param *restrict param);
```

DESCRIPTION

`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数分别用于获取和设置 `attr` 参数中的调度参数属性。`param` 结构体的内容在 `<sched.h>` 头文件中定义。对于 SCHED_FIFO 和 SCHED_RR 策略，`param` 的唯一必需成员是 `sched_priority`。

[TSP] 对于 SCHED_SPORADIC 策略，`param` 结构体的必需成员是 `sched_priority`、`sched_ss_low_priority`、`sched_ss_repl_period`、`sched_ss_init_budget` 和 `sched_ss_max_repl`。指定的 `sched_ss_repl_period` 需要大于或等于指定的 `sched_ss_init_budget` 函数才能成功；如果不是，则函数将失败。`sched_ss_max_repl` 的值必须在包含范围 $[1, \{SS_REPL_MAX\}]$ 内函数才能成功；如果不是，函数将失败。对于 `sched_ss_repl_period` 和 `sched_ss_init_budget` 值是按照此函数提供的方式存储还是四舍五入以与所用时钟的分辨率对齐，这是未指定的。

如果 `pthread_attr_getschedparam()` 或 `pthread_attr_setschedparam()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

`pthread_attr_setschedparam()` 函数应在以下情况失败：

- **ENOTSUP** - 尝试将属性设置为不支持的值。

`pthread_attr_setschedparam()` 函数可能在以下情况失败：

- **EINVAL** - `param` 的值无效。

这些函数不应返回 [EINVAL] 错误代码。

EXAMPLES

无。

APPLICATION USAGE

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前运行的线程。

RATIONALE

如果实现检测到 `pthread_attr_getschedparam()` 或 `pthread_attr_setschedparam()` 的 `attr` 参数指定的值不引用已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_create()`
- `<pthread.h>`
- `<sched.h>`

CHANGE HISTORY

首次发布于 Issue 5

为与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数被标记为 Threads 选项的一部分。
- 为与 IEEE Std 1003.1d-1999 对齐添加了 SCHED_SPORADIC 调度策略。
- 为与 ISO/IEC 9899:1999 标准对齐, 向 `pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 原型添加了 `restrict` 关键字。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/78, 更新了 ERRORS 部分以包含当 `attr` 引用未初始化线程属性对象时的可选错误。

Issue 7

- `pthread_attr_getschedparam()` 和 `pthread_attr_setschedparam()` 函数从 Threads 选项移动到 Base。
- 应用了 Austin Group Interpretation 1003.1-2001 #119, 阐明了 `sched_ss_repl_period` 和 `sched_ss_init_budget` 值的准确性要求。

- 删除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。
 - 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0451 [314]。
-

1.121. pthread_attr_getschedpolicy

SYNOPSIS

```
#include <pthread.h>

int pthread_attr_getschedpolicy(const pthread_attr_t *restrict
                                int *restrict policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int polic
```

DESCRIPTION

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数分别用于获取和设置 `attr` 参数中的 `schedpolicy` 属性。

`policy` 参数支持的值应包括 `SCHED_FIFO`、`SCHED_RR` 和 `SCHED_OTHER`，这些值在 `<sched.h>` 头文件中定义。当使用调度策略 `SCHED_FIFO`、`SCHED_RR` 或 `SCHED_SPORADIC` 执行的线程在等待互斥锁时，当互斥锁解锁时，它们应按优先级顺序获取互斥锁。

如果传递给 `pthread_attr_getschedpolicy()` 或 `pthread_attr_setschedpolicy()` 的 `attr` 参数值不引用已初始化的线程属性对象，则其行为是未定义的。

RETURN VALUE

如果成功，`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

`pthread_attr_setschedpolicy()` 函数在以下情况下应失败：

- **[ENOTSUP]**

尝试将属性设置为不支持的值。

`pthread_attr_setschedpolicy()` 函数在以下情况下可能失败：

- [EINVAL]

`policy` 的值无效。

这些函数不应返回错误代码 [EINTR]。

以下部分为参考信息。

EXAMPLES

无。

APPLICATION USAGE

在设置这些属性后，可以使用 `pthread_create()` 函数创建具有指定属性的线程。使用这些例程不会影响当前正在运行的线程。

有关线程调度属性及其默认设置的更多详细信息，请参见 [2.9.4 Thread Scheduling](#)。

RATIONALE

如果实现检测到传递给 `pthread_attr_getschedpolicy()` 或 `pthread_attr_setschedpolicy()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应该失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getscope()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedparam()`

- `pthread_create()`

XBD `<pthread.h>` , `<sched.h>`

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含在内。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数被标记为线程和线程执行调度选项的一部分。

如果实现不支持线程执行调度选项，不需要提供存根，因此移除了 [ENOSYS] 错误条件。

为与 IEEE Std 1003.1d-1999 对齐，添加了 SCHED_SPORADIC 调度策略。

为与 ISO/IEC 9899:1999 标准对齐，在 `pthread_attr_getschedpolicy()` 原型中添加了 **restrict** 关键字。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/79，在 APPLICATION USAGE 部分添加了对 [2.9.4 Thread Scheduling](#) 的引用。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/80，更新了 ERRORS 部分，为 `attr` 引用未初始化线程属性对象的情况包含了可选错误。

Issue 7

`pthread_attr_getschedpolicy()` 和 `pthread_attr_setschedpolicy()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是 Base 的一部分。

为未初始化线程属性对象的 [EINVAL] 错误被移除；这种情况会导致未定义行为。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0452 [314]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0263 [757]。

1.122. `pthread_attr_getscope`, `pthread_attr_setscope` — 获取和设置争用作用域属性 (REALTIME THREADS)

概要

```
#include <pthread.h>

int pthread_attr_getscope(const pthread_attr_t *restrict attr,
                           int *restrict contentionscope);

int pthread_attr_setscope(pthread_attr_t *attr, int contentionscope);
```

描述

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数分别用于获取和设置 *attr* 对象中的 *contentionscope* (争用作用域) 属性。

contentionscope 属性可以取以下值: `PTHREAD_SCOPE_SYSTEM`，表示系统调度争用作用域；或 `PTHREAD_SCOPE_PROCESS`，表示进程调度争用作用域。符号 `PTHREAD_SCOPE_SYSTEM` 和 `PTHREAD_SCOPE_PROCESS` 在 `<pthread.h>` 头文件中定义。

如果传递给 `pthread_attr_getscope()` 或 `pthread_attr_setscope()` 的 *attr* 参数值不指向已初始化的线程属性对象，则行为是未定义的。

返回值

如果成功，`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数应返回零；否则，应返回错误码以指示错误。

错误

`pthread_attr_setscope()` 函数在以下情况下应失败：

- [ENOTSUP]: 尝试将属性设置为不支持的值。

`pthread_attr_setscope()` 函数在以下情况下可能失败：

- **[EINVAL]**: *contentionscope* 的值无效。

这些函数不应返回 **[EINTR]** 错误码。

示例

无。

应用程序使用

设置这些属性后，可以使用 `pthread_create()` 创建具有指定属性的线程。使用这些例程不会影响当前运行的线程。

有关线程调度属性及其默认设置的更多详细信息，请参阅 [2.9.4 线程调度](#)。

原理

如果实现检测到传递给 `pthread_attr_getscope()` 或 `pthread_attr_setscope()` 的 *attr* 参数值不指向已初始化的线程属性对象，建议函数应该失败并报告 **[EINVAL]** 错误。

未来方向

无。

另请参阅

- `pthread_attr_destroy()`
- `pthread_attr_getinheritsched()`
- `pthread_attr_getschedpolicy()`
- `pthread_attr_getschedparam()`
- `pthread_create()`

XBD `<pthread.h>` , `<sched.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含在内。

标记为实时线程特性组的一部分。

Issue 6

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数被标记为线程和线程执行调度选项的一部分。

由于如果实现不支持线程执行调度选项，则不需要提供存根，因此移除了 `[ENOSYS]` 错误条件。

为了与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getscope()` 原型添加了 **restrict** 关键字。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/81，在应用程序使用部分添加了对 [2.9.4 线程调度](#) 的引用。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/82，更新了错误部分，为 *attr* 指向未初始化线程属性对象的情况包含可选错误。

Issue 7

`pthread_attr_getscope()` 和 `pthread_attr_setscope()` 函数仅标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。

移除了未初始化线程属性对象的 `[EINVAL]` 错误；此条件导致未定义行为。

应用了 POSIX.1-2008，技术更正 1，XSH/TC1-2008/0453 [314]。

应用了 POSIX.1-2008，技术更正 2，XSH/TC2-2008/0264 [757]。

1.123. `pthread_attr_getstack`, `pthread_attr_setstack` — 获取和设置栈属性

概要

```
[TSA TSS]
#include <pthread.h>

int pthread_attr_getstack(const pthread_attr_t *restrict attr,
                          void **restrict stackaddr,
                          size_t *restrict stacksize);

int pthread_attr_setstack(pthread_attr_t *attr,
                         void *stackaddr,
                         size_t stacksize);
```

描述

`pthread_attr_getstack()` 和 `pthread_attr_setstack()` 函数分别用于获取和设置 `attr` 对象中的线程创建栈属性 `stackaddr` 和 `stacksize`。

栈属性指定要用作所创建线程栈的存储区域。存储的基址（最低可寻址字节）应为 `stackaddr`，存储的大小应为 `stacksize` 字节。`stacksize` 必须至少为 `{PTHREAD_STACK_MIN}`。如果 `stackaddr` 不满足实现定义的对齐要求，`pthread_attr_setstack()` 函数可能返回 `[EINVAL]` 错误。由 `stackaddr` 和 `stacksize` 描述的栈内的所有页面必须对线程而言都是可读和可写的。

如果在设置 `stackaddr` 属性之前调用 `pthread_attr_getstack()` 函数，其行为是未指定的。

如果传递给 `pthread_attr_getstack()` 或 `pthread_attr_setstack()` 的 `attr` 参数值不引用已初始化的线程属性对象，则行为是未定义的。

返回值

成功完成后，这些函数应返回 0 值；否则，应返回错误编号以指示错误。

如果成功，`pthread_attr_getstack()` 函数应将栈属性值存储在 `stackaddr` 和 `stacksize` 中。

错误

`pthread_attr_setstack()` 函数应在以下情况下失败：

- **[EINVAL]**

`stacksize` 的值小于 {PTHREAD_STACK_MIN} 或超过实现定义的限制。

`pthread_attr_setstack()` 函数可能在以下情况下失败：

- **[EINVAL]**

`stackaddr` 的值没有适当的对齐以用作栈，或者 `((char *) stackaddr + stacksize)` 缺乏适当的对齐。

- **[EACCES]**

由 `stackaddr` 和 `stacksize` 描述的栈页面对线程而言并非都可读和可写。

这些函数不应返回 [EINTR] 错误代码。

示例

无。

应用程序使用

这些函数适用于线程栈必须放置在内存特定区域的环境中的应用程序。

虽然应用程序似乎可以通过在指定栈区域外提供受保护页面来检测栈溢出，但这无法以可移植的方式完成。实现可以自由地将线程的初始栈指针放置在指定区域内的任何位置，以适应机器的栈指针行为和分配要求。此外，在某些架构上，如 IA-64，“溢出”可能意味着在区域内分配的两个独立栈指针将在区域中间某处重叠。

成功调用 `pthread_attr_setstack()` 后，由 `stackaddr` 参数指定的存储区域由实现控制，如 [2.9.8 应用程序管理的线程栈的使用](#) 中所述。

`stackaddr` 属性的规范存在几个模糊之处，使得无法以可移植的方式使用这些函数。例如，标准允许实现对 `stackaddr` 强加任意的对齐要求。应用程序不能假设从 `malloc()` 获得的缓冲区是适当对齐的。请注意，虽然传递给

`pthread_attr_setstack()` 的 `stacksize` 值必须满足对齐要求，但对于 `pthread_attr_setstacksize()` 则不是这样，后者必须在必要时增加指定的大小以实现适当的对齐。

原理

如果实现检测到传递给 `pthread_attr_getstack()` 或 `pthread_attr_setstack()` 的 `attr` 参数值不引用已初始化的线程属性对象，建议函数应该失败并报告 [EINVAL] 错误。

未来方向

无。

另请参见

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstacksize()`
- `pthread_create()`
- XBD `<limits.h>`
- XBD `<pthread.h>`

变更历史

首次发布于 Issue 6。作为 XSI 选项的一部分开发，并通过 IEEE PASC Interpretation 1003.1 #101 纳入 BASE。

IEEE Std 1003.1-2001/Cor 2-2004

- 应用了项目 XSH/TC2/D6/83，更新了应用程序使用部分以引用 2.9.8 应用程序管理的线程栈的使用。
- 应用了项目 XSH/TC/D6/84，更新了错误部分，为 `attr` 引用未初始化线程属性对象的情况包含可选错误。

Issue 7

- 应用了 SD5-XSH-ERN-66，纠正了 [EINVAL] 错误条件下 `attr` 的使用。
 - 应用了 Austin Group Interpretation 1003.1-2001 #057，阐明了在设置 `stackaddr` 属性之前调用函数时的行为。
 - 应用了 SD5-XSH-ERN-157，更新了应用程序使用部分。
 - 在描述和应用程序使用部分更新了 `stackaddr` 属性的描述。
 - 移除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。
-

1.124. pthread_attr_getstackaddr, pthread_attr_setstackaddr

SYNOPSIS

```
[0B] int pthread_attr_getstackaddr(const pthread_attr_t *restrict
                                     void **restrict stackaddr);
int pthread_attr_setstackaddr(pthread_attr_t *attr, void *stack
```

DESCRIPTION

`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 函数分别用于获取和设置 attr 对象中的线程创建栈地址属性。

栈地址属性指定用于所创建线程栈的存储位置。该存储的大小应至少为 {PTHREAD_STACK_MIN}。

RETURN VALUE

成功完成后，`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 应返回值 0；否则，应返回错误码以指示错误。

如果成功，`pthread_attr_getstackaddr()` 函数会将栈地址属性值存储在 stackaddr 中。

ERRORS

这些函数可能在以下情况下失败：

- **[EINVAL]**
- attr 指定的值不引用已初始化的线程属性对象。

这些函数不应返回错误码 [EINTR]。

EXAMPLES

无。

APPLICATION USAGE

栈地址属性的规范存在若干歧义，使得这些接口的可移植使用成为不可能。将单个地址参数描述为"栈"并未指定地址与该地址所暗示的"栈"之间的特定关系。例如，该地址可以被视为用作栈的缓冲区的低内存地址，也可以被视为用作新线程的初始栈指针寄存器值的地址。这两种情况只有在栈从低内存向高内存"向上"增长，并且"push"操作先在内存中存储值然后递增栈指针寄存器的机器上才是相同的。此外，在栈从高内存向低内存"向下"增长的机器上，将地址解释为"低内存"地址需要确定栈的预期大小。IEEE Std 1003.1-2001 引入了新接口 `pthread_attr_setstack()` 和 `pthread_attr_getstack()` 来解决这些歧义。

在成功调用 `pthread_attr_setstackaddr()` 后，由 `stackaddr` 参数指定的存储区域由实现控制，如"应用程序管理线程栈的使用"中所述。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_attr_getstack()`
- `pthread_attr_getstacksize()`
- `pthread_attr_setstack()`
- `pthread_create()`
- IEEE Std 1003.1-2001 基础定义卷， `<limits.h>` , `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_attr_getstackaddr()` 和 `pthread_attr_setstackaddr()` 函数被标记为线程和线程栈地址属性选项的一部分。

为与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getstackaddr()` 原型添加了 **restrict** 关键字。

这些函数被标记为过时。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/85，更新 APPLICATION USAGE 部分以引用"应用程序管理线程栈的使用"。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/86，更新 ERRORS 部分以包含当 attr 引用未初始化线程属性对象时的可选错误。

1.125. `pthread_attr_getstacksize`, `pthread_attr_setstacksize` — 获取和设置栈大小属性

概要

```
#include <pthread.h>

int pthread_attr_getstacksize(const pthread_attr_t *restrict attr,
                               size_t *restrict stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr, size_t stacksize);
```

描述

`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数分别用于获取和设置 *attr* 对象中的线程创建 *stacksize* (栈大小) 属性。

stacksize 属性定义了为所创建线程的栈分配的最小栈大小 (以字节为单位)。

如果传递给 `pthread_attr_getstacksize()` 或 `pthread_attr_setstacksize()` 的 *attr* 参数值未引用已初始化的线程属性对象，则行为未定义。

返回值

成功完成后，`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 应返回值 0；否则，应返回错误码以指示错误。

如果成功，`pthread_attr_getstacksize()` 函数将 *stacksize* 属性值存储在 *stacksize* 中。

错误

`pthread_attr_setstacksize()` 函数应在以下情况下失败：

[EINVAL]

stacksize 的值小于 {PTHREAD_STACK_MIN} 或超过系统强加的限制。

这些函数不应返回 [EINTR] 错误码。

示例

无。

应用用法

无。

原理

如果实现检测到传递给 `pthread_attr_getstacksize()` 或 `pthread_attr_setstacksize()` 的 *attr* 参数值未引用已初始化的线程属性对象，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

参见

- `pthread_attr_destroy()`
- `pthread_attr_getdetachstate()`
- `pthread_create()`
- `<limits.h>`
- `<pthread.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数被标记为线程和线程栈大小属性选项的一部分。

为与 ISO/IEC 9899:1999 标准对齐，向 `pthread_attr_getstacksize()` 原型添加了 **restrict** 关键字。

应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/43，将 SYNOPSIS 中的边距代码从 TSA 更正为 TSS，并将 CHANGE HISTORY 从"线程栈地址属性"选项更新为"线程栈大小属性"选项。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/87，更新 ERRORS 部分以包含当 *attr* 引用未初始化线程属性对象时的可选错误。

Issue 7

`pthread_attr_getstacksize()` 和 `pthread_attr_setstacksize()` 函数仅被标记为线程栈大小属性选项的一部分，因为线程选项现在是基础的一部分。

删除了未初始化线程属性对象的 [EINVAL] 错误；此条件导致未定义行为。

应用了 POSIX.1-2008，Technical Corrigendum 2，XSH/TC2-2008/0265 [757]。

1.126. pthread_cancel

名称

pthread_cancel — 取消线程的执行

概要

```
#include <pthread.h>

int pthread_cancel(pthread_t thread);
```

描述

`pthread_cancel()` 函数应请求取消 `thread`。目标线程的可取消状态和类型决定了取消操作何时生效。当取消操作被执行时，`thread` 的取消清理处理程序将被调用。当最后一个取消清理处理程序返回时，`thread` 的线程特定数据析构函数将被调用。当最后一个析构函数返回时，`thread` 应被终止。请求取消僵尸线程不应被视为错误。

目标线程中的取消处理应相对于调用线程从 `pthread_cancel()` 返回异步运行。

如果 `thread` 引用使用 `thrd_create()` 创建的线程，则行为是未定义的。

返回值

如果成功，`pthread_cancel()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_cancel()` 函数不应返回 [EINTR] 错误代码。

以下部分是提供信息的。

示例

无。

应用用法

无。

原理

考虑了两种备选函数来向线程发送取消通知。一种是定义一个新的 SIGCANCEL 信号，该信号在传递时具有取消语义；另一种是定义新的 `pthread_cancel()` 函数，该函数会触发取消语义。

新信号的优势在于，传递标准与尝试传递信号时的标准非常相似，因此将取消通知作为信号被视为一致的。实际上，许多实现使用特殊信号来实现取消。另一方面，除了 `pthread_kill()` 之外，没有可以与此信号一起使用的信号函数，并且传递的取消信号的行为与任何先前存在的已定义信号都不相同。

特殊函数的好处包括认识到这个信号是因为相似的传递标准而被定义的，并且这是取消请求和信号之间的唯一共同行为。此外，取消传递机制不必实现为信号。与信号相比，它与语言异常机制有很强的（如果不是更强的话）相似性，如果传递机制在可见上更接近信号，这种相似性可能会被掩盖。

最终，考虑到新信号与现有信号函数的使用有如此多的例外情况，使用信号会产生误导。特殊函数解决了这个问题。这个函数被仔细定义，以便希望在信号之上提供取消函数的实现可以这样做。特殊函数还意味着实现不必使用信号来实现取消。

如果实现在线程生命周期结束后检测到使用线程 ID，建议该函数应该失败并报告 [ESRCH] 错误。

历史上的实现在使用指示僵尸线程的线程 ID 执行 `pthread_cancel()` 时的结果各不相同。一些实现指示成功，无需进一步操作，因为线程已经终止；而另一些实现则给出 [ESRCH] 错误。由于本标准中的线程生命周期定义涵盖僵尸线程，在这种情况下如上所述的 [ESRCH] 错误是不合适的，给出此错误的实现不符合规范。

使用 `pthread_cancel()` 取消使用 `thrd_create()` 创建的线程是未定义的，因为 `thrd_join()` 无法指示线程被取消。标准开发者考虑添加一个 `thrd_canceled` 枚举常量，在这种情况下 `thrd_join()` 将返回该常量。但是，这个返回值在为符合 ISO C 标准而编写的代码中是意外的，并且它也不能解

决仅使用 ISO C `<threads.h>` 接口的线程（例如符合 ISO C 标准的第三方库创建的线程）无法处理被取消的问题，因为 ISO C 标准不提供取消清理处理程序。

未来方向

无。

另请参阅

- `pthread_exit()`
- `pthread_cond_clockwait()`
- `pthread_join()`
- `pthread_setcancelstate()`

XBD `<pthread.h>`

变更历史

第 5 版首次发布。为与 POSIX 线程扩展对齐而包含。

第 6 版

`pthread_cancel()` 函数被标记为线程选项的一部分。

第 7 版

`pthread_cancel()` 函数从线程选项移动到基础部分。

应用 Austin Group 解释 1003.1-2001 #142，移除 [ESRCH] 错误条件。

第 8 版

应用 Austin Group 缺陷 792，添加要求即将僵尸线程的线程 ID 传递给 `pthread_cancel()` 不被视为错误。

应用 Austin Group 缺陷 1302，更新页面以考虑 `<threads.h>` 接口的添加。

1.127. `pthread_cleanup_pop`, `pthread_cleanup_push`

SYNOPSIS

```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

DESCRIPTION

`pthread_cleanup_pop()` 函数应当从调用线程的取消清理栈顶部移除例程，并可选地调用它（如果 *execute* 为非零值）。

`pthread_cleanup_push()` 函数应当将指定的取消清理处理程序 *routine* 推入调用线程的取消清理栈。当以下情况发生时，取消清理处理程序将从取消清理栈中弹出并使用参数 *arg* 调用：

- 线程退出（即调用 `pthread_exit()`）。
- 线程响应取消请求。
- 线程使用非零的 *execute* 参数调用 `pthread_cleanup_pop()`。

未指定 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 是宏还是函数。如果为了访问实际函数而抑制宏定义，或者程序定义了具有这些名称的外部标识符，则行为是未定义的。应用程序应确保它们以语句形式出现，并在相同的词法作用域内成对出现（即，`pthread_cleanup_push()` 宏可以被认为扩展到一个标记列表，其第一个标记是 `'{'`，而 `pthread_cleanup_pop()` 扩展到一个标记列表，其最后一个标记是对应的 `'}'`）。

如果自从填充跳转缓冲区以来有任何未匹配的 `pthread_cleanup_push()` 或 `pthread_cleanup_pop()` 调用，则调用 `longjmp()` 或 `siglongjmp()` 的效果是未定义的。从取消清理处理程序内部调用 `longjmp()` 或 `siglongjmp()` 的效果也是未定义的，除非跳转缓冲区也在取消清理处理程序中填充。

调用取消清理处理程序可能终止任何正在由线程执行的代码块的执行，该代码块的执行在相应的 `pthread_cleanup_push()` 调用之后开始。

使用 `return`、`break`、`continue` 和 `goto` 过早离开由一对 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数调用描述的代码块的效果是未定义的。

RETURN VALUE

`pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数不应返回值。

ERRORS

未定义错误。

这些函数不应返回 [EINTR] 错误码。

EXAMPLES

以下是使用线程原语实现可取消的、写者优先的读写锁的示例：

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
                  wcond;
    int lock_count; /* < 0 .. 由写者持有。 */
                    /* > 0 .. 由 lock_count 个读者持有。 */
                    /* = 0 .. 无人持有。 */
    int waiting_writers; /* 等待的写者数量。 */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
    while ((l->lock_count < 0) || (l->waiting_writers != 0))
        pthread_cond_wait(&l->rcond, &l->lock);
    pthread_cleanup_pop(l);
}
```

```

        pthread_cond_wait(&l->rcond, &l->lock);
        l->lock_count++;
    /*
     * 注意 pthread_cleanup_pop 执行
     * waiting_reader_cleanup。
     */
    pthread_cleanup_pop(1);
}

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
        /*
         * 这只在我们被取消时发生。如果锁没有被写者持有,
         * 可能有读者因为 waiting_writers 为正而被阻塞; 他们现在可以解除
         */
        pthread_cond_broadcast(&l->rcond);
    }
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
    /*
     * 注意 pthread_cleanup_pop 执行
     * waiting_writer_cleanup。
     */
    pthread_cleanup_pop(1);
}

```

```
void
release_write_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->lock_count = 0;
    if (l->waiting_writers == 0)
        pthread_cond_broadcast(&l->rcond);
    else
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

/*
 * 此函数用于初始化读写锁。
 */
void
initialize_rwlock(rwlock *l)
{
    pthread_mutex_init(&l->lock, pthread_mutexattr_default);
    pthread_cond_init(&l->wcond, pthread_condattr_default);
    pthread_cond_init(&l->rcond, pthread_condattr_default);
    l->lock_count = 0;
    l->waiting_writers = 0;
}

reader_thread()
{
    lock_for_read(&lock);
    pthread_cleanup_push(release_read_lock, &lock);
    /*
     * 线程拥有读锁。
     */
    pthread_cleanup_pop(1);
}

writer_thread()
{
    lock_for_write(&lock);
    pthread_cleanup_push(release_write_lock, &lock);
    /*
     * 线程拥有写锁。
     */
    pthread_cleanup_pop(1);
}
```

APPLICATION USAGE

推入和弹出取消清理处理程序的两个例程，`pthread_cleanup_push()` 和 `pthread_cleanup_pop()`，可以被认为是左括号和右括号。它们总是需要匹配。

RATIONALE

推入和弹出取消清理处理程序的两个例程，`pthread_cleanup_push()` 和 `pthread_cleanup_pop()`，必须出现在相同词法作用域的限制允许高效的宏或编译器实现和高效的存储管理。这些例程作为宏的示例实现可能如下所示：

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, **__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler; \
}

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

这些例程的更有野心的实现可能通过允许编译器注意到取消清理处理程序是常量并且可以内联扩展而做得更好。

POSIX.1-2024 当前版本未指定在 POSIX 系统接口函数中执行的信号处理程序中调用 `longjmp()` 的效果。如果实现想要允许此操作并给程序员合理的行为，`longjmp()` 函数必须调用自调用 `setjmp()` 以来已推入但未弹出的所有取消清理处理程序。

考虑一个使用信号的线程调用的多线程函数。如果在 `qsort()` 操作期间向信号处理程序传递信号，并且该处理程序调用 `longjmp()`（这反过来不调用取消清理处理程序），则由 `qsort()` 函数创建的辅助线程不会被取消。相反，它们将继续执行并写入参数数组，即使数组可能已经从栈中弹出。

请注意，指定的清理处理机制特别与 C 语言相关，虽然对表达清理的统一机制的要求是语言无关的，但在其他语言中使用的机制可能完全不同。此外，这种机制实际上只是因为 C 语言缺乏真正的异常机制而必需的，这将是理想的解决方案。

没有取消清理安全函数的概念。如果应用程序在其信号处理程序中没有取消点，在调用异步不安全函数时阻止任何可能包含取消点的信号，或者在调用异步不安全函数时禁用取消，则所有函数都可以安全地从取消清理例程中调用。

FUTURE DIRECTIONS

无。

SEE ALSO

`pthread_cancel()`, `pthread_setcancelstate()`

XBD `<pthread.h>`

CHANGE HISTORY

首次在 Issue 5 中发布。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_cleanup_pop()` 和 `pthread_cleanup_push()` 函数被标记为线程选项的一部分。

添加了 APPLICATION USAGE 部分。

规范性文本已更新，以避免对应用程序要求使用"必须"一词。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/88，更新了 DESCRIPTION 以描述过早离开由 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数定义的代码块的后果。

Issue 7

`pthread_cleanup_pop()` 和 `pthread_cleanup_push()` 函数从线程选项移动到基本 (Base)。

应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0454 [229]。

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0268 [624]。

Issue 8

应用了 Austin Group Defect 613，阐明了自动对象生命周期与取消清理函数的关系。

1.128. `pthread_cleanup_push`, `pthread_cleanup_pop`

概要

```
#include <pthread.h>

void pthread_cleanup_pop(int execute);
void pthread_cleanup_push(void (*routine)(void*), void *arg);
```

描述

`pthread_cleanup_pop()` 函数应当移除调用线程的取消清理堆栈顶部的例程，并可选择性地调用它（如果 `execute` 参数非零）。

`pthread_cleanup_push()` 函数应当将指定的取消清理处理程序 `routine` 推入调用线程的取消清理堆栈中。取消清理处理程序应当在以下情况下从取消清理堆栈中弹出，并使用参数 `arg` 调用：

- 线程退出（即调用 `pthread_exit()`）。
- 线程响应取消请求。
- 线程使用非零的 `execute` 参数调用 `pthread_cleanup_pop()`。

未指定 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 是宏还是函数。如果为了访问实际函数而抑制宏定义，或者程序定义了具有这些名称的外部标识符，则行为是未定义的。应用程序应确保它们作为语句出现，并在相同的词法作用域内成对出现（即，`pthread_cleanup_push()` 宏可以看作扩展为一个以 '{' 开头的标记列表，而 `pthread_cleanup_pop()` 扩展为以对应 '}' 结尾的标记列表）。

如果在填充跳转缓冲区后进行了任何 `pthread_cleanup_push()` 或 `pthread_cleanup_pop()` 调用，而没有相应的匹配调用，则调用 `longjmp()` 或 `siglongjmp()` 的效果是未定义的。从取消清理处理程序内部调用 `longjmp()` 或 `siglongjmp()` 的效果也是未定义的，除非跳转缓冲区也在取消清理处理程序中填充。

调用取消清理处理程序可能会终止线程执行的任何代码块的执行，该代码块的执行是在相应的 `pthread_cleanup_push()` 调用之后开始的。

使用 `return`、`break`、`continue` 和 `goto` 过早离开由一对 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数调用描述的代码块的效果是未定义的。

返回值

`pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数不应返回任何值。

错误

未定义任何错误。

这些函数不应返回 [EINTR] 错误代码。

示例

以下示例使用线程原语实现可取消的、写者优先的读写锁：

```
typedef struct {
    pthread_mutex_t lock;
    pthread_cond_t rcond,
                  wcond;
    int lock_count; /* < 0 .. 由写者持有。 */
                    /* > 0 .. 由 lock_count 个读者持有。 */
                    /* = 0 .. 无人持有。 */
    int waiting_writers; /* 等待的写者数量。 */
} rwlock;

void
waiting_reader_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_read(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    pthread_cleanup_push(waiting_reader_cleanup, l);
```

```

        while ((l->lock_count < 0) || (l->waiting_writers != 0))
            pthread_cond_wait(&l->rcond, &l->lock);
        l->lock_count++;
        /*
         * 注意 pthread_cleanup_pop 执行
         * waiting_reader_cleanup。
         */
        pthread_cleanup_pop(1);
    }

void
release_read_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    if (--l->lock_count == 0)
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

void
waiting_writer_cleanup(void *arg)
{
    rwlock *l;

    l = (rwlock *) arg;
    if ((--l->waiting_writers == 0) && (l->lock_count >= 0)) {
        /*
         * 这只在我们被取消时发生。如果
         * 锁不由写者持有，可能有读者因为
         * waiting_writers 为正而被阻塞；他们
         * 现在可以被解除阻塞。
         */
        pthread_cond_broadcast(&l->rcond);
    }
    pthread_mutex_unlock(&l->lock);
}

void
lock_for_write(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->waiting_writers++;
    pthread_cleanup_push(waiting_writer_cleanup, l);
    while (l->lock_count != 0)
        pthread_cond_wait(&l->wcond, &l->lock);
    l->lock_count = -1;
    /*
     * 注意 pthread_cleanup_pop 执行
     * waiting_writer_cleanup。
     */
}

```

```
        */
        pthread_cleanup_pop(1);
    }

void
release_write_lock(rwlock *l)
{
    pthread_mutex_lock(&l->lock);
    l->lock_count = 0;
    if (l->waiting_writers == 0)
        pthread_cond_broadcast(&l->rcond);
    else
        pthread_cond_signal(&l->wcond);
    pthread_mutex_unlock(&l->lock);
}

/*
 * 这个函数被调用来初始化读写锁。
 */
void
initialize_rwlock(rwlock *l)
{
    pthread_mutex_init(&l->lock, pthread_mutexattr_default);
    pthread_cond_init(&l->wcond, pthread_condattr_default);
    pthread_cond_init(&l->rcond, pthread_condattr_default);
    l->lock_count = 0;
    l->waiting_writers = 0;
}

reader_thread()
{
    lock_for_read(&lock);
    pthread_cleanup_push(release_read_lock, &lock);
    /*
     * 线程拥有读锁。
     */
    pthread_cleanup_pop(1);
}

writer_thread()
{
    lock_for_write(&lock);
    pthread_cleanup_push(release_write_lock, &lock);
    /*
     * 线程拥有写锁。
     */
    pthread_cleanup_pop(1);
}
```

应用程序用法

推入和弹出取消清理处理程序的两个例程，`pthread_cleanup_push()` 和 `pthread_cleanup_pop()`，可以看作是左括号和右括号。它们总是需要匹配。

基本原理

推入和弹出取消清理处理程序的两个例程 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 必须出现在相同词法作用域的限制，允许高效的宏或编译器实现和高效的存储管理。这些例程作为宏的示例实现可能如下所示：

```
#define pthread_cleanup_push(rtn,arg) { \
    struct _pthread_handler_rec __cleanup_handler, ***__head; \
    __cleanup_handler.rtn = rtn; \
    __cleanup_handler.arg = arg; \
    (void) pthread_getspecific(_pthread_handler_key, &__head); \
    __cleanup_handler.next = *__head; \
    *__head = &__cleanup_handler; \
}

#define pthread_cleanup_pop(ex) \
    *__head = __cleanup_handler.next; \
    if (ex) (*__cleanup_handler.rtn)(__cleanup_handler.arg); \
}
```

这些例程的更雄心勃勃的实现可能会通过允许编译器注意到取消清理处理程序是常量并且可以内联扩展而做得更好。

POSIX.1-2024 当前版本未指定从在 POSIX 系统接口函数中执行的信号处理程序调用 `longjmp()` 的效果。如果实现想要允许这一点并给程序员合理的行为，`longjmp()` 函数必须调用自调用 `setjmp()` 以来已推入但未弹出的所有取消清理处理程序。

考虑一个由使用信号的线程调用的多线程函数。如果在 `qsort()` 操作期间将信号传递给信号处理程序，并且该处理程序调用 `longjmp()`（这又不会调用取消清理处理程序），那么由 `qsort()` 函数创建的辅助线程将不会被取消。相反，它们将继续执行并写入参数数组，即使数组可能已从堆栈中弹出。

请注意，指定的清理处理机制特别与 C 语言绑定，虽然对表达清理的统一机制的要求是语言无关的，但在其他语言中使用的机制可能完全不同。此外，这种机制实际上仅因为 C 语言缺乏真正的异常机制而必需，而异常机制将是理想的解决方案。

没有取消清理安全函数的概念。如果应用程序在其信号处理程序中没有取消点，在调用异步不安全函数时阻塞其处理程序可能有取消点的任何信号，或者在调用异步不安全函数时禁用取消，那么所有函数都可以从取消清理例程中安全调用。

未来方向

无。

另请参阅

`pthread_cancel()` ,
`pthread_setcancelstate()`

XBD `<pthread.h>`

更改历史

首次发布于 Issue 5。为了与 POSIX 线程扩展保持一致而包含。

Issue 6

`pthread_cleanup_pop()` 和 `pthread_cleanup_push()` 函数被标记为线程选项的一部分。

添加了应用程序用法部分。

规范性文本已更新，避免对应用程序要求使用"必须"一词。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/88，更新描述以描述过早离开由 `pthread_cleanup_push()` 和 `pthread_cleanup_pop()` 函数定义的代码块的后果。

Issue 7

`pthread_cleanup_pop()` 和 `pthread_cleanup_push()` 函数从线程选项移至基础。

应用了 POSIX.1-2008，技术勘误表 1，XSH/TC1-2008/0454 [229]。

应用了 POSIX.1-2008，技术勘误表 2，XSH/TC2-2008/0268 [624]。

Issue 8

应用了 Austin Group 缺陷 613，阐明了自动对象生命周期与取消清理函数的关系。

1.129. `pthread_cond_broadcast`, `pthread_cond_signal` — 广播或信号通知条件变量

概要

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

描述

这些函数应解除在条件变量上阻塞的线程。

`pthread_cond_broadcast()` 函数应作为单个原子操作，确定是否有线程在指定的条件变量 `cond` 上阻塞，并解除所有这些线程的阻塞。

`pthread_cond_signal()` 函数应作为单个原子操作，确定是否有线程在指定的条件变量 `cond` 上阻塞，并解除至少一个这些线程的阻塞。

如果有多个线程在条件变量上阻塞，调度策略应决定线程解除阻塞的顺序。当每个因 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 而解除阻塞的线程从其对 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的调用中返回时，该线程应拥有其在调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 时所关联的互斥锁。被解除阻塞的线程应根据调度策略（如果适用）竞争互斥锁，并且就像每个线程都调用了 `pthread_mutex_lock()` 一样。

调用 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 函数的线程无论当前是否拥有调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的线程在等待期间与条件变量关联的互斥锁；然而，如果需要可预测的调度行为，那么调用 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的线程应锁定该互斥锁。

如果 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 确定没有线程在 `cond` 上阻塞，则这些函数应不起作用。

如果传递给 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的 `cond` 参数值不引用已初始化的条件变量，则行为未定义。

返回值

如果成功，`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

这些函数不应返回 `[EINTR]` 错误码。

示例

无。

应用使用

当共享变量状态以多个线程可以继续其任务的方式发生更改时，使用 `pthread_cond_broadcast()` 函数。考虑单个生产者/多个消费者问题，其中生产者可以在多个消费者一次访问一个项目的列表上插入多个项目。通过调用 `pthread_cond_broadcast()` 函数，生产者会通知所有可能正在等待的消费者，从而应用程序在多处理器上获得更多的吞吐量。此外，`pthread_cond_broadcast()` 使实现读写锁更容易。当写者释放其锁时，需要 `pthread_cond_broadcast()` 函数来唤醒所有等待的读者。最后，两阶段提交算法可以使用此广播函数通知所有客户端即将到来的事务提交。

在异步调用的信号处理程序中使用 `pthread_cond_signal()` 函数是不安全的。即使它是安全的，仍然存在布尔值的 `pthread_cond_wait()` 测试之间的竞争条件，而这种竞争无法被有效地消除。

因此，互斥锁和条件变量不适合通过在信号处理程序中运行的代码发信号来释放等待的线程。

原理说明

如果实现检测到传递给 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的 `cond` 参数值不引用已初始化的条件变量，建议

函数应失败并报告 `[EINVAL]` 错误。

条件信号的多次唤醒

在多处理器上，`pthread_cond_signal()` 的实现可能无法避免解除在条件变量上阻塞的多个线程的阻塞。例如，考虑以下 `pthread_cond_wait()` 和 `pthread_cond_signal()` 的部分实现，按给定顺序由两个线程执行。一个线程正在尝试等待条件变量，另一个线程正在并发执行 `pthread_cond_signal()`，而第三个线程已经在等待。

```
pthread_cond_wait(mutex, cond):
    value = cond->value;                      /* 1 */
    pthread_mutex_unlock(mutex);                /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) {                 /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex);                 /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++;                  /* 4 */
    if (cond->waiter) {             /* 5 */
        sleeper = cond->waiter;      /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper);        /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

结果是多个线程可以因其对 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的调用而返回，这是由于一次对 `pthread_cond_signal()` 的调用。这种效果被称为"虚假唤醒"。注意这种情况是自纠正的，因为被这样唤醒的线程数量是有限的；例如，在上述事件序列之后，下一个调用 `pthread_cond_wait()` 的线程会阻塞。

虽然这个问题可以解决，但对于仅很少发生的边缘条件的效率损失是不可接受的，特别是考虑到无论如何必须检查与条件变量关联的谓词。纠正这个问题会不必要地减少这个用于所有更高级别同步操作的基本构建块的并发程度。

允许虚假唤醒的另一个好处是应用程序被迫围绕条件等待编写谓词测试循环。这也使应用程序能够容忍可能在应用程序的其他部分编码的同一条件变量上的

多余条件广播或信号。因此，产生的应用程序更加健壮。因此，POSIX.1-2024 明确记录了虚假唤醒可能发生。

未来方向

无。

另请参见

- [pthread_cond_clockwait\(\)](#)
- [pthread_cond_destroy\(\)](#)

XBD 4.15.2 内存同步，

更改历史

在第5期中首次发布。

包含用于与 POSIX 线程扩展对齐。

第6期

`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数被标记为线程选项的一部分。
添加了应用使用部分。

第7期

`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数从线程选项移动到基础规范。
删除了未初始化条件变量的 `[EINVAL]` 错误；此条件导致未定义行为。

第8期

应用 Austin Group 缺陷 609，添加原子性要求。

应用 Austin Group 缺陷 1216，添加 `pthread_cond_clockwait()`。

1.130. pthread_cond_destroy, pthread_cond_init

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *restrict cond,
                      const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

DESCRIPTION

`pthread_cond_destroy()` 函数应销毁由 `cond` 指向的给定条件变量；该对象实际上变为未初始化状态。实现可能导致 `pthread_cond_destroy()` 将 `cond` 引用的对象设置为无效值。被销毁的条件变量对象可以使用 `pthread_cond_init()` 重新初始化；在对象被销毁后以其他方式引用该对象的结果是未定义的。

销毁没有线程当前阻塞在其上的已初始化条件变量是安全的。尝试销毁其他线程当前阻塞在其上的条件变量会导致未定义行为。

`pthread_cond_init()` 函数应使用由 `attr` 引用的属性初始化由 `cond` 引用的条件变量。如果 `attr` 为 NULL，则应使用默认条件变量属性；效果与传递默认条件变量属性对象的地址相同。成功初始化后，条件变量的状态应变为已初始化状态。

关于进一步要求，请参阅 2.9.9 同步对象副本和替代映射。

尝试初始化已初始化的条件变量会导致未定义行为。

在默认条件变量属性合适的情况下，可以使用宏 PTHREAD_COND_INITIALIZER 来初始化条件变量。效果应等同于通过调用参数 `attr` 指定为 NULL 的 `pthread_cond_init()` 进行动态初始化，只是不执行错误检查。

如果 `pthread_cond_destroy()` 的 `cond` 参数指定的值不引用已初始化的条件变量，则行为未定义。

如果 `pthread_cond_init()` 的 `attr` 参数指定的值不引用已初始化的条件变量属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_cond_destroy()` 和 `pthread_cond_init()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS

如果出现以下情况，`pthread_cond_init()` 函数应失败：

- **[EAGAIN]**

系统缺乏初始化另一个条件变量所需的必要资源（内存除外）。

- **[ENOMEM]**

内存不足以初始化条件变量。

这些函数不应返回错误代码 [EINTR]。

EXAMPLES

条件变量可以在所有阻塞在其上的线程被唤醒后立即销毁。例如，考虑以下代码：

```
struct list {
    pthread_mutex_t lm;
    ...
};

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
};

/* 查找列表元素并预留它。 */
struct elt *
list_find(struct list *lp, key k)
{
```

```

    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
    while ((ep = find_elt(l, k) != NULL) && ep->busy)
        pthread_cond_wait(&ep->notbusy, &lp->lm);
    if (ep != NULL)
        ep->busy = 1;
    pthread_mutex_unlock(&lp->lm);
    return(ep);
}

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... 从列表中移除 ep ...
    ep->busy = 0; /* 防御性编程。 */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&ep->notbusy);
    free(ep);
}

```

在此示例中，条件变量及其列表元素可以在所有等待它的线程被唤醒后（行 A）立即释放（行 B），因为互斥锁和代码确保没有其他线程可以接触要删除的元素。

APPLICATION USAGE

无。

RATIONALE

如果实现检测到 `pthread_cond_destroy()` 的 `cond` 参数指定的值不引用已初始化的条件变量，建议函数应失败并报告 [EINVAL] 错误。

如果实现检测到 `pthread_cond_destroy()` 或 `pthread_cond_init()` 的 `cond` 参数指定的值引用了正在被其他线程使用的条件变量（例如，在 `pthread_cond_wait()` 调用中），或者检测到 `pthread_cond_init()` 的 `cond` 参数指定的值引用了已初始化的条件变量，建议函数应失败并报告 [EBUSY] 错误。

如果实现检测到 `pthread_cond_init()` 的 `attr` 参数指定的值不引用已初始化的条件变量属性对象，建议函数应失败并报告 [EINVAL] 错误。

另请参阅 `pthread_mutex_destroy()`。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_cond_broadcast()`
- `pthread_cond_clockwait()`
- `pthread_mutex_destroy()`

XBD

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_cond_destroy()` 和 `pthread_cond_init()` 函数被标记为线程选项的一部分。

应用了 IEEE PASC Interpretation 1003.1c #34，更新了 DESCRIPTION。

为了与 ISO/IEC 9899:1999 标准对齐，在 `pthread_cond_init()` 原型中添加了 **restrict** 关键字。

Issue 7

`pthread_cond_destroy()` 和 `pthread_cond_init()` 函数从线程选项移至基础。

移除了未初始化条件变量和未初始化条件变量属性对象的 [EINVAL] 错误；此条件会导致未定义行为。

移除了已使用条件变量或已初始化条件变量的 [EBUSY] 错误；此条件会导致未定义行为。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0455 [70]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0269 [972] 和 XSH/TC2-2008/0270 [910]。

1.131. pthread_cond_destroy, pthread_cond_init

— 销毁和初始化条件变量

概要

```
#include <pthread.h>

int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_init(pthread_cond_t *restrict cond,
                     const pthread_condattr_t *restrict attr);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

描述

`pthread_cond_destroy()` 函数应销毁由 `cond` 指定的给定条件变量；该对象实际上变为未初始化状态。实现可能导致 `pthread_cond_destroy()` 将 `cond` 引用的对象设置为无效值。已销毁的条件变量对象可以使用 `pthread_cond_init()` 重新初始化；在销毁后以其他方式引用该对象的结果是未定义的。

销毁一个没有线程当前被阻塞的已初始化条件变量是安全的。尝试销毁其他线程当前被阻塞的条件变量会导致未定义行为。

`pthread_cond_init()` 函数应使用 `attr` 引用的属性初始化由 `cond` 引用的条件变量。如果 `attr` 为 NULL，应使用默认的条件变量属性；效果与传递默认条件变量属性对象的地址相同。成功初始化后，条件变量的状态应变为已初始化。

有关进一步要求，请参阅 [2.9.9 同步对象副本和替代映射](#)。

尝试初始化已初始化的条件变量会导致未定义行为。

在默认条件变量属性合适的情况下，可以使用宏 `PTHREAD_COND_INITIALIZER` 初始化条件变量。效果应等同于通过调用 `pthread_cond_init()` 并将参数 `attr` 指定为 NULL 进行动态初始化，只是不执行错误检查。

如果 `pthread_cond_destroy()` 的 `cond` 参数指定的值不引用已初始化的条件变量，则行为是未定义的。

如果 `pthread_cond_init()` 的 `attr` 参数指定的值不引用已初始化的条件变量属性对象，则行为是未定义的。

返回值

如果成功，`pthread_cond_destroy()` 和 `pthread_cond_init()` 函数应返回零；否则应返回错误号以指示错误。

错误

如果出现以下情况，`pthread_cond_init()` 函数应失败：

- **[EAGAIN]**

系统缺乏必要的资源（内存除外）来初始化另一个条件变量。

- **[ENOMEM]**

内存不足，无法初始化条件变量。

这些函数不应返回 **[EINTR]** 错误代码。

示例

条件变量可以在所有被阻塞的线程被唤醒后立即销毁。例如，考虑以下代码：

```
struct list {
    pthread_mutex_t lm;
    ...
};

struct elt {
    key k;
    int busy;
    pthread_cond_t notbusy;
    ...
};

/* 查找列表元素并保留它。 */
struct elt *
list_find(struct list *lp, key k)
{
    struct elt *ep;

    pthread_mutex_lock(&lp->lm);
```

```

        while ((ep = find_elt(l, k) != NULL) && ep->busy)
            pthread_cond_wait(&ep->notbusy, &lp->lm);
        if (ep != NULL)
            ep->busy = 1;
        pthread_mutex_unlock(&lp->lm);
        return(ep);
    }

delete_elt(struct list *lp, struct elt *ep)
{
    pthread_mutex_lock(&lp->lm);
    assert(ep->busy);
    ... 从列表中移除 ep ...
    ep->busy = 0; /* 谨慎处理。 */
(A) pthread_cond_broadcast(&ep->notbusy);
    pthread_mutex_unlock(&lp->lm);
(B) pthread_cond_destroy(&ep->notbusy);
    free(ep);
}

```

在此示例中，条件变量及其列表元素可以在所有等待它的线程被唤醒（A 行）后立即释放（B 行），因为互斥锁和代码确保没有其他线程可以触摸要删除的元素。

应用程序使用

无。

原理

如果实现检测到 `pthread_cond_destroy()` 的 `cond` 参数指定的值不引用已初始化的条件变量，建议函数应失败并报告 **[EINVAL]** 错误。

如果实现检测到 `pthread_cond_destroy()` 或 `pthread_cond_init()` 的 `cond` 参数指定的值引用正在被另一个线程使用的条件变量（例如，在 `pthread_cond_wait()` 调用中），或者检测到 `pthread_cond_init()` 的 `cond` 参数指定的值引用已初始化的条件变量，建议函数应失败并报告 **[EBUSY]** 错误。

如果实现检测到 `pthread_cond_init()` 的 `attr` 参数指定的值不引用已初始化的条件变量属性对象，建议函数应失败并报告 **[EINVAL]** 错误。

另请参阅 `pthread_mutex_destroy()`。

未来方向

无。

参见

- `pthread_cond_broadcast()`
- `pthread_cond_clockwait()`
- `pthread_mutex_destroy()`
- XBD `<pthread.h>`

变更历史

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_cond_destroy()` 和 `pthread_cond_init()` 函数被标记为线程选项的一部分。
- 应用了 IEEE PASC Interpretation 1003.1c #34，更新了描述。
- 为了与 ISO/IEC 9899:1999 标准对齐，在 `pthread_cond_init()` 原型中添加了 **restrict** 关键字。

Issue 7

- `pthread_cond_destroy()` 和 `pthread_cond_init()` 函数从线程选项移动到基础。
- 删除了针对未初始化条件变量和未初始化条件变量属性对象的 **[EINVAL]** 错误；此条件导致未定义行为。
- 删除了针对已使用或已初始化条件变量的 **[EBUSY]** 错误；此条件导致未定义行为。
- 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0455 [70]。

- 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0269 [972] 和 XSH/TC2-2008/0270 [910]。

1.132. `pthread_cond_broadcast`, `pthread_cond_signal` — 广播或信号通知条件

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_signal(pthread_cond_t *cond);
```

DESCRIPTION

这些函数应解除阻塞在条件变量上的线程。

`pthread_cond_broadcast()` 函数应作为单个原子操作，确定是否有线程阻塞在指定的条件变量 `cond` 上，并解除所有这些线程的阻塞。

`pthread_cond_signal()` 函数应作为单个原子操作，确定是否有线程阻塞在指定的条件变量 `cond` 上，并解除至少一个这些线程的阻塞。

如果有多个线程阻塞在条件变量上，调度策略应决定解除线程阻塞的顺序。当每个因 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 而解除阻塞的线程从其 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用返回时，该线程应拥有其调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 时关联的互斥锁。被解除阻塞的线程应根据调度策略（如果适用）竞争互斥锁，如同每个线程都调用了 `pthread_mutex_lock()`。

`pthread_cond_broadcast()` 或 `pthread_cond_signal()` 函数可由线程调用，无论该线程当前是否拥有调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的线程在等待期间与条件变量关联的互斥锁；但是，如果需要可预测的调度行为，那么调用 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的线程应锁定该互斥锁。

如果 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 确定没有线程阻塞在 `cond` 上，这些函数应不起作用。

如果传递给 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的 `cond` 参数值不引用已初始化的条件变量，则行为未定义。

RETURN VALUE

如果成功，`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

这些函数不应返回 `[EINTR]` 错误码。

EXAMPLES

无。

APPLICATION USAGE

当共享变量状态以多个线程可以继续执行其任务的方式发生改变时，使用 `pthread_cond_broadcast()` 函数。考虑单生产者/多消费者问题，其中生产者可以在消费者一次访问一个项目的列表上插入多个项目。通过调用 `pthread_cond_broadcast()` 函数，生产者会通知所有可能正在等待的消费者，从而在多处理器上获得更高的吞吐量。此外，`pthread_cond_broadcast()` 使得实现读写锁更容易。当写者释放其锁时，需要 `pthread_cond_broadcast()` 函数来唤醒所有等待的读者。最后，两阶段提交算法可以使用此广播函数通知所有客户端即将到来的事务提交。

在异步调用的信号处理程序中使用 `pthread_cond_signal()` 函数是不安全的。即使它是安全的，在布尔 `pthread_cond_wait()` 的测试之间仍存在无法有效消除的竞争条件。

因此，互斥锁和条件变量不适合通过在信号处理程序中运行的代码发信号来释放等待的线程。

RATIONALE

如果实现检测到传递给 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的 `cond` 参数值不引用已初始化的条件变量，建议函数应失败并报告 `[EINVAL]` 错误。

条件信号的多次唤醒

在多处理器上，`pthread_cond_signal()` 的实现可能无法避免解除阻塞多个阻塞在条件变量上的线程。例如，考虑以下 `pthread_cond_wait()` 和 `pthread_cond_signal()` 的部分实现，由两个线程按给定顺序执行。一个线程试图等待条件变量，另一个线程并发执行 `pthread_cond_signal()`，而第三个线程已经在等待。

```
pthread_cond_wait(mutex, cond):
    value = cond->value;          /* 1 */
    pthread_mutex_unlock(mutex);   /* 2 */
    pthread_mutex_lock(cond->mutex); /* 10 */
    if (value == cond->value) {    /* 11 */
        me->next_cond = cond->waiter;
        cond->waiter = me;
        pthread_mutex_unlock(cond->mutex);
        unable_to_run(me);
    } else
        pthread_mutex_unlock(cond->mutex); /* 12 */
    pthread_mutex_lock(mutex);      /* 13 */

pthread_cond_signal(cond):
    pthread_mutex_lock(cond->mutex); /* 3 */
    cond->value++;                /* 4 */
    if (cond->waiter) {           /* 5 */
        sleeper = cond->waiter;    /* 6 */
        cond->waiter = sleeper->next_cond; /* 7 */
        able_to_run(sleeper);      /* 8 */
    }
    pthread_mutex_unlock(cond->mutex); /* 9 */
```

结果是多个线程可能从其 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中返回，这是由于一次 `pthread_cond_signal()` 调用造成的。这种效应称为"伪唤醒"。请注意，这种情况是自我修正的，因为被这样唤醒的线程数量是有限的；例如，在上述事件序列之后，下一个调用 `pthread_cond_wait()` 的线程会被阻塞。

虽然这个问题可以被解决，但对于仅偶尔发生的边缘条件损失的效率是不可接受的，特别是考虑到无论如何都必须检查与条件变量关联的谓词。纠正此问题会不必要地降低这个所有高级同步操作基本构建块的并发度。

允许伪唤醒的一个额外好处是应用程序被迫在条件等待周围编码谓词测试循环。这也使应用程序能够容忍应用程序其他部分可能编码的同一条件变量上的多余条件广播或信号。因此，产生的应用程序更加健壮。因此，POSIX.1-2024 明确记录了可能发生伪唤醒。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- XBD 4.15.2 内存同步
- `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5

包含用于与 POSIX 线程扩展对齐。

Issue 6

`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数被标记为线程选项的一部分。

添加了 APPLICATION USAGE 部分。

Issue 7

`pthread_cond_broadcast()` 和 `pthread_cond_signal()` 函数从线程选项移动到基础。

删除了未初始化条件变量的 `[EINVAL]` 错误；此条件导致未定义行为。

Issue 8

应用 Austin Group 缺陷 609，添加了原子性要求。

应用 Austin Group 缺陷 1216，添加了 `pthread_cond_clockwait()`。

1.133. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` — 等待条件变量

SYNOPSIS (概要)

```
#include <pthread.h>

int pthread_cond_clockwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           clockid_t clock_id,
                           const struct timespec *restrict absti

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict absti

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

DESCRIPTION (描述)

`pthread_cond_clockwait()`、`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数应该在条件变量上阻塞。应用程序应确保这些函数在调用线程锁定 `mutex` 的情况下被调用；否则，会导致错误（对于 PTHREAD_MUTEX_ERRORCHECK 和健壮互斥锁）或未定义行为（对于其他互斥锁）。

这些函数原子地释放 `mutex` 并使调用线程在条件变量 `cond` 上阻塞；这里的"原子地"是指"相对于另一个线程对互斥锁然后对条件变量的访问是原子的"。也就是说，如果另一个线程在即将阻塞的线程释放互斥锁后能够获取该互斥锁，那么该线程中后续的 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 调用的行为应等同于在即将阻塞的线程已经阻塞之后发出。

成功返回时，互斥锁应该已被锁定并由调用线程拥有。

如果 `mutex` 是一个健壮互斥锁，其持有者在持有锁时终止且状态可恢复，那么即使函数返回 [EOWNERDEAD]，互斥锁也应被获取。

使用条件变量时，总是有一个与每个条件等待相关的涉及共享变量的布尔谓词，如果线程应该继续执行，该谓词为真。`pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 函数可能会发生虚假唤醒。由于从这些函数返回并不意味着该谓词的值有任何意义，因此在这样的返回时应重新评估谓词。

当线程在条件变量上等待时，已为 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 操作指定了特定的互斥锁，在该互斥锁和条件变量之间形成了一个动态绑定，只要至少有一个线程在该条件变量上阻塞，该绑定就保持有效。在此期间，任何线程尝试使用不同的互斥锁在该条件变量上等待的效果是未定义的。一旦所有等待的线程都被解除阻塞（如通过 `pthread_cond_broadcast()` 操作），该条件变量上的下一个等待操作应与该等待操作指定的互斥锁形成新的动态绑定。尽管在线程从条件变量等待中被解除阻塞到返回给调用者或开始取消清理之间的时间内，条件变量和互斥锁之间的动态绑定可能被移除或替换，但被解除阻塞的线程应始终重新获取在条件等待操作调用中指定的互斥锁。

条件等待（无论是否超时）是一个取消点。当线程的可取消性类型设置为 `PTHREAD_CANCEL_DEFERRED` 时，在条件等待期间处理取消请求的一个副作用是互斥锁在调用第一个取消清理处理程序之前（实际上）被重新获取。效果如同线程被解除阻塞，允许执行到从 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用返回的点，但在该点注意到取消请求，而不是返回给调用者，而是开始线程取消活动，包括调用取消清理处理程序。

在调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 时被阻塞而被取消的线程，如果有其他线程在该条件变量上阻塞，则不应消耗可能同时指向该条件变量的任何条件信号。

`pthread_cond_clockwait()` 函数应等同于 `pthread_cond_wait()`，不同的是，如果以 `clock_id` 指示的时钟测量的 `abstime` 指定的绝对时间已过（即，该时钟测量的当前时间等于或超过 `abstime`）在条件 `cond` 被信号或广播之前，或者在调用时 `abstime` 指定的绝对时间已经过去，则返回错误。实现应支持将 `CLOCK_REALTIME` 和 `CLOCK_MONOTONIC` 作为 `clock_id` 参数传递给 `pthread_cond_clockwait()`。当发生此类超时时，`pthread_cond_clockwait()` 仍会释放并重新获取 `mutex` 引用的互斥锁，并且可能消耗指向该条件变量的条件信号。

`pthread_cond_timedwait()` 函数应等同于 `pthread_cond_clockwait()`，但它缺少 `clock_id` 参数。用于测量 `abstime` 的时钟应来自条件变量的时钟属性，该属性可以在条件变量创建之前通过 `pthread_condattr_setclock()` 设置。如果未设置时钟属性，默认应为 `CLOCK_REALTIME`。

如果信号被传递给正在等待条件变量的线程，从信号处理程序返回后，线程将恢复等待条件变量，如同未被中断一样，或者由于虚假唤醒而返回零。

如果传递给这些函数的 `cond` 或 `mutex` 参数的值分别没有引用已初始化的条件变量或已初始化的互斥锁对象，则行为未定义。

RETURN VALUE (返回值)

除了 [ETIMEDOUT]、[ENOTRECOVERABLE] 和 [EOWNERDEAD] 外，所有这些错误检查都应如同在函数处理开始时立即执行一样，并且应导致错误返回，实际上是在修改 `mutex` 指定的互斥锁或 `cond` 指定的条件变量的状态之前。

成功完成时，应返回零值；否则，应返回错误编号以指示错误。

ERRORS (错误)

这些函数可能失败，如果：

- **[EAGAIN]**

互斥锁是健壮互斥锁，并且系统可用的健壮互斥锁资源将被超出。

- **[ENOTRECOVERABLE]**

互斥锁保护的状态不可恢复。

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，包含先前拥有线程的进程在持有互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

- **[EPERM]**

互斥锁类型是 PTHREAD_MUTEX_ERRORCHECK 或互斥锁是健壮互斥锁，且当前线程不拥有互斥锁。

`pthread_cond_clockwait()` 和 `pthread_cond_timedwait()` 函数可能失败，如果：

- **[ETIMEDOUT]**

`abstime` 指定的时间已经过去。

- **[EINVAL]**

`abstime` 参数指定的纳秒值小于零或大于等于 1000 百万，或者传递给 `pthread_cond_clockwait()` 的 `clock_id` 参数无效或不受支持。

这些函数可能失败，如果：

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，先前的拥有线程在持有互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

假设非零返回值为错误的应用程序需要更新以使用健壮互斥锁，因为线程获取保护当前不一致状态的互斥锁的有效返回是 [EOWNERDEAD]。不检查错误返回的应用程序（由于排除了此类错误出现的可能性）不应使用健壮互斥锁。如果应用程序应该与普通和健壮互斥锁一起工作，它应该检查所有返回值的错误条件，并在必要时采取适当措施。

RATIONALE (原理)

如果实现检测到传递给 `pthread_cond_clockwait()`、
`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的 `cond` 参数值没有引用已初始化的条件变量，或者检测到 `mutex` 参数值没有引用已初始化的互斥锁对象，建议函数应该失败并报告 [EINVAL] 错误。

Condition Wait Semantics (条件等待语义)

需要注意的是，当 `pthread_cond_clockwait()`、
`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 无错误返回时，相关的谓词可能仍然为假。类似地，当 `pthread_cond_clockwait()` 或 `pthread_cond_timedwait()` 因超时错误返回时，相关的谓词可能为真，这是由于超时到期和谓词状态变化之间不可避免的竞争条件。

应用程序需要在任何返回时重新检查谓词，因为它不能确定是否有另一个线程在等待线程来处理信号，如果没有，那么信号就丢失了。检查谓词的责任在于应用程序。

一些实现，特别是在多处理器上，有时可能在条件变量在不同处理器上同时被信号时导致多个线程被唤醒。

一般来说，无论条件等待何时返回，线程都必须重新评估与条件等待相关的谓词，以确定它是否可以安全继续、应该再次等待，还是应该声明超时。从等待返回并不意味着相关的谓词要么为真要么为假。

因此，建议条件等待包含在检查谓词的等效"while 循环"中。

Timed Wait Semantics (超时等待语义)

为超时参数指定绝对时间度量的原因有两个。首先，相对时间度量可以在指定绝对时间的函数之上轻松实现，但在指定相对超时的函数之上指定绝对超时存在竞争条件。例如，假设 `clock_gettime()` 返回当前时间，`cond_relative_timed_wait()` 使用相对超时：

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

如果线程在第一个语句和最后一个语句之间被抢占，线程将阻塞太久。然而，如果使用绝对超时，阻塞是无关紧要的。如果在循环中多次使用绝对超时，也不需要重新计算。

对于操作员不连续地提前系统时钟的情况，预期实现会处理在中间时间到期的任何超时等待，如同该时间实际发生一样。

Choice of Clock (时钟选择)

在超时等待时，应谨慎决定哪个时钟最合适。`pthread_cond_timedwait()` 默认使用的系统时钟 CLOCK_REALTIME 可能为了与实际时间校正而向前和向后跳变。CLOCK_MONOTONIC 保证不会向后跳变，并且也必须在实时中前进，因此通过 `pthread_cond_clockwait()` 或 `pthread_condattr_setclock()` 使用它可能更合适。

Cancellation and Condition Wait (取消和条件等待)

条件等待，无论是否超时，都是一个取消点。也就是说，`pthread_cond_clockwait()`、`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数是注意到挂起（或并发）取消请求的点。这样做的原因是在这些点可能发生无限期等待——无论等待什么事件，即使程序完全正确，也可能永远不会发生；例如，等待的某些输入数据可能永远不会被发送。通过使条件等待成为取消点，线程可以被取消并执行其取消清理处理程序，即使它可能被卡在某个无限期等待中。

在条件变量上阻塞时处理取消请求的一个副作用是在调用任何取消清理处理程序之前重新获取互斥锁。这样做是为了确保取消清理处理程序在与条件等待函数调用之前和之后的关键代码相同的状态中执行。在从 Ada 或 C++ 等语言与 POSIX 线程接口时，也需要这个规则，这些语言可能选择将取消映射到语言异常；这个规则确保每个保护关键段的异常处理程序总是可以安全地依赖于相关互斥锁已被锁定的事实，无论异常在关键段内的确切位置被引发。没有这个规则，异常处理程序就没有统一的规则来遵循关于锁的行为，因此编码将变得非常繁琐。

因此，由于必须就取消在等待期间传递时锁的状态做出某些声明，选择了使应用程序编码最方便和无错误的定义。

在线程在条件变量上阻塞时处理取消请求时，实现需要确保如果有其他线程在该条件变量上等待，线程不会消耗任何指向该条件变量的条件信号。指定这个规则是为了避免如果这两个独立请求（一个作用于线程，另一个作用于条件变量）不被独立处理时可能发生的死锁条件。

Performance of Mutexes and Condition Variables（互斥锁和条件变量的性能）

互斥锁预期只被锁定几条指令。这种做法几乎被程序员避免长串行执行区域（这将减少总的有效并行性）的愿望自动强制执行。

使用互斥锁和条件变量时，人们试图确保通常的情况是锁定互斥锁、访问共享数据、解锁互斥锁。在条件变量上等待应该是一个相对罕见的情况。例如，在实现读写锁时，获取读锁的代码通常只需要（在互斥下）增加读取器计数并返回。只有在已经有活动写入器时，调用线程才会在条件变量上实际等待。所以同步操作的效率以互斥锁锁定/解锁的成本为界限，而不是以条件等待为界限。注意，在通常情况下没有上下文切换。

这并不是说条件等待的效率不重要。由于每次 Ada 会合至少需要一次上下文切换，在条件变量上等待的效率很重要。在条件变量上等待的成本应该略多于上下文切换的最小成本加上解锁和锁定互斥锁的时间。

Features of Mutexes and Condition Variables（互斥锁和条件变量的特性）

有人建议将互斥锁的获取和释放与条件等待解耦。这被拒绝了，因为操作的组合性质实际上促进了实时实现。这些实现可以以对调用者透明的方式原子地在条件变量和互斥锁之间移动高优先级线程。这可以防止额外的上下文切换，并在等待线程被信号时提供更确定的互斥锁获取。因此，公平性和优先级问题可以通过调度规则直接处理。此外，当前的条件等待操作与现有实践相匹配。

Scheduling Behavior of Mutexes and Condition Variables

(互斥锁和条件变量的调度行为)

试图通过指定排序规则来干扰调度策略的同步原语被认为是不受欢迎的。等待互斥锁和条件变量的线程被选择继续的顺序取决于调度策略，而不是某个固定顺序（例如，FIFO 或优先级）。因此，调度策略决定哪些线程被唤醒并允许继续。

Timed Condition Wait (超时条件等待)

`pthread_cond_clockwait()` 和 `pthread_cond_timedwait()` 函数允许应用程序在给定时间后放弃等待特定条件。示例如下：

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_clockwait(&t.cond, &t.mn,
                                    CLOCK_MONOTONIC, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

通过使超时参数为绝对值，不需要在程序每次检查其阻塞谓词时重新计算。如果超时是相对的，则必须在每次调用之前重新计算。这将特别困难，因为这样的代码需要考虑由于在谓词为真或超时到期之前发生的额外广播或条件变量信号导致的额外唤醒的可能性。使用 `CLOCK_MONOTONIC` 而不是 `CLOCK_REALTIME` 意味着超时不受系统时钟更改的影响。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (另请参见)

- `pthread_cond_broadcast()`
- XBD 4.15.2 Memory Synchronization, `<pthread.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 5。包含用于与 POSIX 线程扩展对齐。

Issue 6

`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数被标记为 Threads 选项的一部分。

应用了 The Open Group Corrigendum U021/9，更正了 `pthread_cond_wait()` 函数的原型。

DESCRIPTION 通过添加 Clock Selection 选项的语义进行更新，以与 IEEE Std 1003.1j-2000 对齐。

响应 IEEE PASC Interpretation 1003.1c #28，ERRORS 部分为 [EPERM] 添加了额外情况。

为与 ISO/IEC 9899:1999 标准对齐，向 `pthread_cond_timedwait()` 和 `pthread_cond_wait()` 原型添加了 `restrict` 关键字。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/89，更新 DESCRIPTION 以与 `pthread_cond_destroy()` 函数保持一致，该函数声明销毁没有线程当前阻塞的已初始化条件变量是安全的。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/90，将 DESCRIPTION 中的措辞从 "the cancelability enable state" 更新为 "the cancelability type"。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/91，更新 ERRORS 部分，从 `pthread_cond_wait()` 函数中移除与 `abstime` 相关的错误情况，并使与 `abstime` 相关的错误情况对 `pthread_cond_timedwait()` 强制执行，以与其他函数保持一致。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/92，向 RATIONALE 部分添加新段落，说明应用程序应检查此函数的任何返回时的谓词。

Issue 7

应用了 SD5-XSH-ERN-44，更改了 [EINVAL] 错误的"shall fail"情况的定义。

从 The Open Group Technical Standard, 2006, Extended API Set Part 3 进行了更改。

`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数从 Threads 选项移动到 Base。

移除了未初始化条件变量或未初始化互斥锁对象的 [EINVAL] 错误；这种情况导致未定义行为。

[EPERM] 错误被修订并移动到 `pthread_cond_timedwait()` 函数的错误条件的"shall fail"列表。

更新 DESCRIPTION 以阐明当 `mutex` 是健壮互斥锁时的行为。

更新 ERRORS 部分以包含 PTHREAD_MUTEX_ERRORCHECK 互斥锁的"shall fail"情况。

重写 DESCRIPTION 以阐明未定义行为仅发生在未强制要求 [EPERM] 错误的互斥锁上。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0456 [91,286,437] 和 XSH/TC1-2008/0457 [239]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0271 [749]。

Issue 8

应用了 Austin Group Defect 354，添加了 [EAGAIN] 错误。

应用了 Austin Group Defect 1162，将 "an error code" 更改为 "[EOWNERDEAD]"。

应用了 Austin Group Defects 1216 和 1485，添加了 `pthread_cond_clockwait()`。

1.134. `pthread_cond_clockwait`, `pthread_cond_timedwait`, `pthread_cond_wait` — 等待条件变量

SYNOPSIS

```
#include <pthread.h>

int pthread_cond_clockwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           clockid_t clock_id,
                           const struct timespec *restrict absti

int pthread_cond_timedwait(pthread_cond_t *restrict cond,
                           pthread_mutex_t *restrict mutex,
                           const struct timespec *restrict absti

int pthread_cond_wait(pthread_cond_t *restrict cond,
                      pthread_mutex_t *restrict mutex);
```

DESCRIPTION

`pthread_cond_clockwait()`、`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数应该在条件变量上阻塞。应用程序应确保这些函数在调用线程锁定 `mutex` 的情况下被调用；否则，会导致错误（对于 PTHREAD_MUTEX_ERRORCHECK 和健壮互斥锁）或未定义行为（对于其他互斥锁）。

这些函数原子性地释放 `mutex` 并导致调用线程在条件变量 `cond` 上阻塞；这里的“原子性”是指“相对于其他线程对互斥锁然后是条件变量的访问而言是原子的”。也就是说，如果另一个线程能够在即将阻塞的线程释放互斥锁后获取它，那么该线程中随后调用 `pthread_cond_broadcast()` 或 `pthread_cond_signal()` 的行为应该就像在即将阻塞的线程阻塞后发出一样。

成功返回时，互斥锁应该已被锁定并且应由调用线程拥有。

如果 `mutex` 是一个健壮互斥锁，其中持有锁的所有者终止且状态可恢复，即使函数返回 [OWNERDEAD]，互斥锁也应该被获取。

使用条件变量时，总是有一个涉及共享变量的布尔谓词与每个条件等待相关联，如果线程应该继续执行，则该谓词为真。从 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 函数可能发生虚假唤醒。由于从这些函数返回并不意味着该谓词值的任何信息，因此在这样的返回时应该重新评估谓词。

当线程在条件变量上等待时，已为 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 操作指定了特定的互斥锁，在该互斥锁和条件变量之间形成动态绑定，只要至少有一个线程在条件变量上阻塞，该绑定就保持有效。在此期间，任何线程尝试使用不同的互斥锁在该条件变量上等待的效果是未定义的。一旦所有等待线程都被解除阻塞（如通过 `pthread_cond_broadcast()` 操作），该条件变量上的下一个等待操作应与该等待操作指定的互斥锁形成新的动态绑定。即使在线程从条件变量等待中被解除阻塞到它返回给调用者或开始取消清理之间的时间内，条件变量和互斥锁之间的动态绑定可能被移除或替换，被解除阻塞的线程应始终重新获取在其返回的条件等待操作调用中指定的互斥锁。

条件等待（无论是否定时）是一个取消点。当线程的可取消性类型设置为 `PTHREAD_CANCEL_DEFERRED` 时，在条件等待期间处理取消请求的一个副作用是在调用第一个取消清理处理程序之前互斥锁被（实际上）重新获取。效果就像线程被解除阻塞，允许执行到从 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用返回的点上，但在该点注意到取消请求，并且不是返回给调用者，而是开始线程取消活动，这包括调用取消清理处理程序。

因为被取消而在调用 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 时被阻塞的线程已被解除阻塞，如果有其他线程在该条件变量上阻塞，则该线程不应消耗可能同时指向该条件变量的任何条件信号。

`pthread_cond_clockwait()` 函数应等效于 `pthread_cond_wait()`，除了如果以 `clock_id` 指示的时钟测量的 `abstime` 指定的绝对时间在条件 `cond` 被发信号或广播之前过去（即，该时钟测量的当前时间等于或超过 `abstime`），或者在调用时 `abstime` 指定的绝对时间已经过去，则返回错误。实现应支持将 `CLOCK_REALTIME` 和 `CLOCK_MONOTONIC` 作为 `clock_id` 参数传递给 `pthread_cond_clockwait()`。当发生此类超时时，`pthread_cond_clockwait()` 仍会释放并重新获取 `mutex` 引用的互斥锁，并且可能消耗同时指向该条件变量的条件信号。

`pthread_cond_timedwait()` 函数应等效于 `pthread_cond_clockwait()`，除了它缺少 `clock_id` 参数。测量 `abstime` 的时钟应来自条件变量的时钟属性，该属性可以在条件变量创建之前

通过 `pthread_condattr_setclock()` 设置。如果未设置时钟属性，默认应为 `CLOCK_REALTIME`。

如果信号被传递给正在等待条件变量的线程，从信号处理程序返回时，线程恢复等待条件变量，就像它没有被中断一样，或者它应由于虚假唤醒而返回零。

如果这些函数的 `cond` 或 `mutex` 参数指定的值分别未引用已初始化的条件变量或已初始化的互斥锁对象，则行为未定义。

RETURN VALUE

除了 `[ETIMEDOUT]`、`[ENOTRECOVERABLE]` 和 `[EOWNERDEAD]` 外，所有这些错误检查应表现为在函数处理开始时立即执行，并且应导致错误返回，实际上在修改 `mutex` 指定的互斥锁或 `cond` 指定的条件变量状态之前。

成功完成时，应返回零值；否则，应返回错误号以指示错误。

ERRORS

这些函数应在以下情况失败：

- **[EAGAIN]**
- 互斥锁是健壮互斥锁，并且可用的健壮互斥锁拥有的系统资源将被超出。
- **[ENOTRECOVERABLE]**
- 互斥锁保护的状态不可恢复。
- **[EOWNERDEAD]**
- 互斥锁是健壮互斥锁，包含先前拥有线程的进程在持有互斥锁时终止。互斥锁应由调用线程获取，并且由新所有者负责使状态保持一致。
- **[EPERM]**
- 互斥锁类型是 `PTHREAD_MUTEX_ERRORCHECK` 或互斥锁是健壮互斥锁，并且当前线程不拥有互斥锁。

`pthread_cond_clockwait()` 和 `pthread_cond_timedwait()` 函数应在以下情况失败：

- **[ETIMEDOUT]**
- `abstime` 指定的时间已过去。
- **[EINVAL]**

- `abstime` 参数指定了小于零或大于等于 1000 百万的纳秒值，或者传递给 `pthread_cond_clockwait()` 的 `clock_id` 参数无效或不受支持。

这些函数可能在以下情况失败：

- **[EOWNERDEAD]**
- 互斥锁是健壮互斥锁，先前的拥有线程在持有互斥锁时终止。互斥锁应由调用线程获取，并且由新所有者负责使状态保持一致。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES

无。

APPLICATION USAGE

已假定非零返回值是错误的应用程序需要更新以与健壮互斥锁一起使用，因为线程获取保护当前不一致状态的互斥锁的有效返回是 [EOWNERDEAD]。由于排除了此类错误出现的可能性而不检查错误返回的应用程序不应使用健壮互斥锁。如果应用程序应该与普通互斥锁和健壮互斥锁一起工作，它应该检查所有返回值的错误条件，并在必要时采取适当的行动。

RATIONALE

如果实现检测到 `pthread_cond_clockwait()`、
`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 的 `cond` 参数值未引用已初始化的条件变量，或者检测到 `mutex` 参数值未引用已初始化的互斥锁对象，建议函数应该失败并报告 [EINVAL] 错误。

Condition Wait Semantics

需要注意的是，当 `pthread_cond_clockwait()`、
`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 无错误返回时，相关的谓词可能仍然为假。类似地，当 `pthread_cond_clockwait()` 或 `pthread_cond_timedwait()` 因超时错误返回时，由于超时过期和谓词状态变化之间不可避免的竞争，相关的谓词可能为真。

应用程序需要在任何返回时重新检查谓词，因为它不能确定是否有另一个线程正在等待该线程来处理信号，如果没有，则信号会丢失。应用程序有责任检查

谓词。

一些实现，特别是在多处理器上，当条件变量在不同处理器上同时发信号时，有时可能导致多个线程被唤醒。

一般来说，每当条件等待返回时，线程必须重新评估与条件等待相关的谓词，以确定它是否可以安全地继续，应该再次等待，还是应该声明超时。从等待返回并不意味着相关谓词为真或为假。

因此，建议将条件等待包含在检查谓词的等效"while 循环"中。

Timed Wait Semantics

为超时参数指定绝对时间测量有两个原因。首先，相对时间测量可以在指定绝对时间的函数上轻松实现，但在指定相对超时的函数上指定绝对超时存在竞争条件。例如，假设 `clock_gettime()` 返回当前时间，`cond_relative_timed_wait()` 使用相对超时：

```
clock_gettime(CLOCK_REALTIME, &now)
reltime = sleep_til_this_absolute_time - now;
cond_relative_timed_wait(c, m, &reltime);
```

如果线程在第一条语句和最后一条语句之间被抢占，线程会阻塞太长时间。但是，如果使用绝对超时，阻塞是无关紧要的。如果绝对超时在循环中多次使用，例如包围条件等待的循环，也不需要重新计算。

对于系统时钟被操作员不连续提前的情况，预期实现处理任何在中间时间到期的定时等待，就像该时间实际发生一样。

Choice of Clock

在带超时等待时应注意决定哪个时钟最合适。默认情况下 `pthread_cond_timedwait()` 使用的系统时钟 CLOCK_REALTIME 可能为了与实际时间校正而向前和向后跳跃。CLOCK_MONOTONIC 保证不会向后跳跃，并且也必须在实时中前进，因此通过 `pthread_cond_clockwait()` 或 `pthread_condattr_setclock()` 使用它可能更合适。

Cancellation and Condition Wait

条件等待，无论是否定时，都是一个取消点。也就是说，`pthread_cond_clockwait()`、`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数是注意到待处理（或并发）取消请求的点。这样做的原因是在这些点上可能发生无限期等待——无论正在等待什么事件，即使程序完全正确，也可能永远不会发生；例如，正在等待的一些输入数据可能永

远不会被发送。通过使条件等待成为取消点，即使线程可能陷入某种无限期等待，也可以被取消并执行其取消清理处理程序。

当线程在条件变量上被阻塞时处理取消请求的副作用是在调用任何取消清理处理程序之前重新获取互斥锁。这样做是为了确保取消清理处理程序在与位于条件等待函数调用之前和之后的关键代码相同的状态中执行。在从 Ada 或 C++ 等语言与 POSIX 线程接口时，也需要此规则，这些语言可能选择将取消映射到语言异常；此规则确保保护关键部分的每个异常处理程序始终可以安全地依赖于相关互斥锁已被锁定的事实，无论异常在关键部分内的确切位置被引发。没有此规则，异常处理程序就没有统一的规则可以遵循关于锁定的规则，因此编码会变得非常繁琐。

因此，由于必须在取消在等待期间传递时关于锁的状态做出某些声明，因此选择了使应用程序编码最方便和无错误的定义。

当线程在条件变量上被阻塞时处理取消请求，实现必须确保如果有其他线程在该条件变量上等待，线程不消耗指向该条件变量的任何条件信号。指定此规则是为了避免如果这两个独立请求（一个作用于线程，另一个作用于条件变量）没有独立处理时可能发生的死锁条件。

Performance of Mutexes and Condition Variables

互斥锁预期只锁定几条指令。这种做法几乎被程序员避免长串行执行区域的愿望自动强制执行（这会减少总的有效并行性）。

使用互斥锁和条件变量时，人们试图确保通常的情况是锁定互斥锁、访问共享数据并解锁互斥锁。在条件变量上等待应该是相对罕见的情况。例如，当实现读写锁时，获取读锁的代码通常只需要在互斥下增加读取器计数并返回。调用线程只会在已经有活动写入器时才实际在条件变量上等待。因此，同步操作的效率受互斥锁锁定/解锁的成本限制，而不是受条件等待的限制。注意，在通常情况下没有上下文切换。

这并不是说条件等待的效率不重要。由于每次 Ada 会合至少需要一个上下文切换，在条件变量上等待的效率是重要的。在条件变量上等待的成本应该略多于上下文切换的最小成本加上解锁和锁定互斥锁的时间。

Features of Mutexes and Condition Variables

有人建议将互斥锁获取和释放与条件等待解耦。这被拒绝，因为正是操作的组合性质实际上促进了实时实现。这些实现可以透明地将高优先级线程在条件变量和互斥锁之间原子性地移动，这对调用者是透明的。这可以防止额外的上下文切换，并在等待线程被发信号时提供更确定性的互斥锁获取。因此，公平性和优先级问题可以通过调度规则直接处理。此外，当前的条件等待操作符合现有实践。

Scheduling Behavior of Mutexes and Condition Variables

试图通过指定排序规则干扰调度策略的同步原语被认为是不受欢迎的。在互斥锁和条件变量上等待的线程被选择继续执行的顺序依赖于调度策略，而不是某个固定顺序（例如，FIFO 或优先级）。因此，调度策略决定哪些线程被唤醒并允许继续。

Timed Condition Wait

`pthread_cond_clockwait()` 和 `pthread_cond_timedwait()` 函数允许应用程序在给定时间后放弃等待特定条件。示例如下：

```
(void) pthread_mutex_lock(&t.mn);
    t.waiters++;
    clock_gettime(CLOCK_MONOTONIC, &ts);
    ts.tv_sec += 5;
    rc = 0;
    while (! mypredicate(&t) && rc == 0)
        rc = pthread_cond_clockwait(&t.cond, &t.mn,
                                    CLOCK_MONOTONIC, &ts);
    t.waiters--;
    if (rc == 0 || mypredicate(&t))
        setmystate(&t);
(void) pthread_mutex_unlock(&t.mn);
```

通过使超时参数绝对，程序每次检查其阻塞谓词时都不需要重新计算它。如果超时是相对的，就必须在每次调用之前重新计算。这将是特别困难的，因为这样的代码需要考虑在谓词为真或超时到期之前发生的额外广播或条件变量信号导致的额外唤醒的可能性。使用 `CLOCK_MONOTONIC` 而不是 `CLOCK_REALTIME` 意味着超时不受系统时钟更改的影响。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_cond_broadcast()`
- XBD 4.15.2 Memory Synchronization, `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_cond_timedwait()` 和 `pthread_cond_wait()` 函数被标记为 Threads 选项的一部分。

应用 The Open Group Corrigendum U021/9，更正了 `pthread_cond_wait()` 函数的原型。

DESCRIPTION 通过添加 Clock Selection 选项的语义更新，以与 IEEE Std 1003.1j-2000 对齐。

ERRORS 部分有额外情况

1.135. `pthread_condattr_destroy`, `pthread_condattr_init`

SYNOPSIS

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

DESCRIPTION

`pthread_condattr_destroy()` 函数应销毁一个条件变量属性对象；该对象实际上变为未初始化状态。实现可能导致 `pthread_condattr_destroy()` 将 attr 引用的对象设置为无效值。已销毁的 attr 属性对象可以使用 `pthread_condattr_init()` 重新初始化；在对象被销毁后以其他方式引用该对象的结果是未定义的。

`pthread_condattr_init()` 函数应使用实现定义的所有属性的默认值初始化条件变量属性对象 attr。

如果在调用 `pthread_condattr_init()` 时指定了一个已经初始化的 attr 属性对象，则结果未定义。

在条件变量属性对象已被用于初始化一个或多个条件变量后，任何影响该属性对象的函数（包括销毁）都不应影响任何先前已初始化的条件变量。

本卷 POSIX.1-2024 要求两个属性：**clock**（时钟）属性和 **process-shared**（进程共享）属性。

其他属性、它们的默认值以及获取和设置这些属性值的关联函数的名称由实现定义。

如果传递给 `pthread_condattr_destroy()` 的 attr 参数值不引用已初始化的条件变量属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS

`pthread_condattr_init()` 函数在以下情况下应失败：

- [ENOMEM]
- 用于初始化条件变量属性对象的内存不足。

这些函数不应返回 [EINTR] 错误码。

以下各节为提供信息的部分。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

为条件变量定义 **process-shared** (进程共享) 属性的原因与为互斥锁定义该属性的原因相同。

如果实现检测到传递给 `pthread_condattr_destroy()` 的 attr 参数值不引用已初始化的条件变量属性对象，建议该函数应失败并报告 [EINVAL] 错误。

另请参见 `pthread_attr_destroy()` 和 `pthread_mutex_destroy()`。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_attr_destroy()`

- `pthread_cond_destroy()`
- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`
- XBD

CHANGE HISTORY

首次在 Issue 5 中发布。为了与 POSIX 线程扩展对齐而包含在内。

Issue 6

`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数被标记为 Threads (线程) 选项的一部分。

Issue 7

`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数从 Threads (线程) 选项移动到 Base (基础) 中。

对于未初始化的条件变量属性对象的 [EINVAL] 错误被移除；这种情况会导致未定义行为。

信息文本结束。

1.136. `pthread_condattr_getclock`, `pthread_condattr_setclock`

SYNOPSIS (概要)

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict
                               clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                               clockid_t clock_id);
```

DESCRIPTION (描述)

`pthread_condattr_getclock()` 函数应从 *attr* 引用的属性对象中获取 *clock* 属性的值。

`pthread_condattr_setclock()` 函数应在 *attr* 引用的已初始化属性对象中设置 *clock* 属性。如果使用引用 CPU 时间时钟的 *clock_id* 参数调用 `pthread_condattr_setclock()`，该调用应失败。

clock 属性是用于测量 `pthread_cond_timedwait()` 超时服务的时钟的时钟 ID。 *clock* 属性的默认值应引用系统时钟。 *clock* 属性对 `pthread_cond_clockwait()` 函数没有影响。

如果 `pthread_condattr_getclock()` 或 `pthread_condattr_setclock()` 的 *attr* 参数指定的值不引用已初始化的条件变量属性对象，则行为未定义。

RETURN VALUE (返回值)

如果成功，`pthread_condattr_getclock()` 函数应返回零，并将 *attr* 的时钟属性值存储到 *clock_id* 参数引用的对象中。否则，应返回错误号以指示错误。

如果成功，`pthread_condattr_setclock()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS (错误)

`pthread_condattr_setclock()` 函数可能失败的情况：

- [EINVAL]
- *clock_id* 指定的值不引用已知时钟，或者是 CPU 时间时钟。

这些函数不应返回 [EINTR] 错误码。

以下章节为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

无。

RATIONALE (基本原理)

如果实现检测到 `pthread_condattr_getclock()` 或 `pthread_condattr_setclock()` 的 *attr* 参数指定的值不引用已初始化的条件变量属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- `pthread_condattr_destroy()`

- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`
- XBD `<pthread.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 6。源自 IEEE Std 1003.1j-2000。

Issue 7

`pthread_condattr_getclock()` 和 `pthread_condattr_setclock()` 函数从时钟选择选项移至基础规范。

移除了针对未初始化条件变量属性对象的 [EINVAL] 错误；此条件会导致未定义行为。

Issue 8

应用 Austin Group 缺陷 1216，添加了 `pthread_cond_clockwait()`。

1.137. `pthread_condattr_destroy`, `pthread_condattr_init`

概要

```
#include <pthread.h>

int pthread_condattr_destroy(pthread_condattr_t *attr);
int pthread_condattr_init(pthread_condattr_t *attr);
```

描述

`pthread_condattr_destroy()` 函数应销毁一个条件变量属性对象；该对象实际上变为未初始化状态。实现可以导致 `pthread_condattr_destroy()` 将 `attr` 引用的对象设置为无效值。已销毁的 `attr` 属性对象可以使用 `pthread_condattr_init()` 重新初始化；在对象被销毁后以其他方式引用该对象的结果是未定义的。

`pthread_condattr_init()` 函数应使用实现定义的所有属性的默认值初始化条件变量属性对象 `attr`。

如果在调用 `pthread_condattr_init()` 时指定了一个已经初始化的 `attr` 属性对象，则结果未定义。

在条件变量属性对象已用于初始化一个或多个条件变量后，任何影响该属性对象的函数（包括销毁）都不应影响任何先前已初始化的条件变量。

POSIX.1-2024 本卷要求两个属性：**时钟** 属性和 **进程共享** 属性。

附加属性、它们的默认值以及用于获取和设置这些属性值的关联函数的名称是由实现定义的。

如果传递给 `pthread_condattr_destroy()` 的 `attr` 参数的值不引用已初始化的条件变量属性对象，则行为未定义。

返回值

如果成功，`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数应返回零；否则，应返回错误码以指示错误。

错误

`pthread_condattr_init()` 函数可能在以下情况下失败：

- **[ENOMEM]**
- 内存不足，无法初始化条件变量属性对象。

这些函数不应返回 **[EINTR]** 错误码。

以下章节为信息性内容。

示例

无。

应用用法

无。

原理

为条件变量定义 **进程共享** 属性的原因与为互斥锁定义该属性的原因相同。

如果实现检测到传递给 `pthread_condattr_destroy()` 的 `attr` 参数的值不引用已初始化的条件变量属性对象，建议函数应失败并报告 **[EINVAL]** 错误。

另见 `pthread_attr_destroy()` 和 `pthread_mutex_destroy()`。

未来方向

无。

参见

`pthread_attr_destroy()` ,
`pthread_condattr_getpshared()` ,
`pthread_mutex_destroy()`

`pthread_cond_destroy()` ,
`pthread_create()` ,

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数被标记为线程选项的一部分。

Issue 7

`pthread_condattr_destroy()` 和 `pthread_condattr_init()` 函数从线程选项移动到基础规范。

对于未初始化的条件变量属性对象的 **[EINVAL]** 错误被移除；此条件导致未定义行为。

信息性文本结束。

1.138. `pthread_condattr_getclock`, `pthread_condattr_setclock`

概要

```
#include <pthread.h>

int pthread_condattr_getclock(const pthread_condattr_t *restrict
                               clockid_t *restrict clock_id);

int pthread_condattr_setclock(pthread_condattr_t *attr,
                               clockid_t clock_id);
```

描述

`pthread_condattr_getclock()` 函数应当从 `attr` 引用的属性对象中获取 `clock` 属性的值。

`pthread_condattr_setclock()` 函数应当在 `attr` 引用的已初始化属性对象中设置 `clock` 属性。如果使用引用 CPU 时间时钟的 `clock_id` 参数调用 `pthread_condattr_setclock()`，则该调用应当失败。

`clock` 属性是用于测量 `pthread_cond_timedwait()` 超时服务的时钟的时钟 ID。 `clock` 属性的默认值应当引用系统时钟。 `clock` 属性对 `pthread_cond_clockwait()` 函数没有影响。

如果传递给 `pthread_condattr_getclock()` 或 `pthread_condattr_setclock()` 的 `attr` 参数值没有引用已初始化的条件变量属性对象，则行为是未定义的。

返回值

如果成功，`pthread_condattr_getclock()` 函数应当返回零，并将 `attr` 的时钟属性值存储到 `clock_id` 参数引用的对象中。否则，应当返回错误码以指示错误。

如果成功，`pthread_condattr_setclock()` 函数应当返回零；否则，应当返回错误码以指示错误。

错误

`pthread_condattr_setclock()` 函数可能失败，如果：

- `[EINVAL]`
- `clock_id` 指定的值没有引用已知时钟，或者是 CPU 时间时钟。

这些函数不应返回 `[EINTR]` 错误码。

以下章节是提供信息的。

示例

无。

应用程序用法

无。

原理

如果实现检测到传递给 `pthread_condattr_getclock()` 或 `pthread_condattr_setclock()` 的 `attr` 参数值没有引用已初始化的条件变量属性对象，建议该函数应当失败并报告 `[EINVAL]` 错误。

未来方向

无。

另请参见

- `pthread_cond_clockwait()`
- `pthread_cond_destroy()`
- `pthread_condattr_destroy()`

- `pthread_condattr_getpshared()`
- `pthread_create()`
- `pthread_mutex_destroy()`

XBD `<pthread.h>`

变更历史

首次发布于 Issue 6。派生自 IEEE Std 1003.1j-2000。

Issue 7

`pthread_condattr_getclock()` 和 `pthread_condattr_setclock()` 函数从时钟选择选项移至基础部分。

删除了针对未初始化条件变量属性对象的 `[EINVAL]` 错误；此条件导致未定义行为。

Issue 8

应用了 Austin Group 缺陷 1216，添加了 `pthread_cond_clockwait()`。

信息文本结束。

1.139. pthread_create - 线程创建

概要

```
#include <pthread.h>

int pthread_create(pthread_t *restrict thread,
                  const pthread_attr_t *restrict attr,
                  void *(*start_routine)(void*),
                  void *restrict arg);
```

描述

`pthread_create()` 函数应在进程中创建一个具有 `attr` 指定属性的新线程。如果 `attr` 为 `NULL`，则应使用默认属性。如果 `attr` 指定的属性在后续被修改，线程的属性不会受到影响。成功完成后，`pthread_create()` 应将创建的线程的 ID 存储在 `thread` 引用的位置。

线程被创建时执行 `start_routine`，以 `arg` 作为其唯一参数。如果 `start_routine` 返回，效果应如同隐式调用了 `pthread_exit()`，使用 `start_routine` 的返回值作为退出状态。注意，最初调用 `main()` 的线程与此不同。当它从 `main()` 返回时，效果应如同隐式调用了 `exit()`，使用 `main()` 的返回值作为退出状态。

信号状态初始化

新线程的信号状态应按如下方式初始化：

- 信号掩码应从创建线程继承。
- 新线程的待处理信号集合应为空。

线程局部的当前本地化设置 [XSI] 和备用栈不会被继承。

浮点环境应从创建线程继承。

如果 `pthread_create()` 失败，不会创建新线程，`thread` 引用的位置内容是未定义的。

[TCT] 如果定义了 `_POSIX_THREAD_CPUTIME`，新线程应有一个可访问的 CPU 时间时钟，该时钟的初始值应设置为零。

如果 `pthread_create()` 的 `attr` 参数指定的值不是指向已初始化的线程属性对象，则行为是未定义的。

返回值

如果成功，`pthread_create()` 函数应返回零；否则，应返回错误号以指示错误。

错误

`pthread_create()` 函数应在以下情况下失败：

`[EAGAIN]`

系统缺乏创建另一个线程所需的必要资源，或者系统强制的进程内线程总数限制 `{PTHREAD_THREADS_MAX}` 将被超过。

`[EPERM]`

调用者没有适当的权限来设置所需的调度参数或调度策略。

`pthread_create()` 函数不应返回 `[EINTR]` 错误码。

示例

无。

应用程序使用

实现没有要求在新创建的线程开始执行之前，创建线程的 ID 就必须可用。调用线程可以通过 `pthread_create()` 函数的 `thread` 参数获取创建线程的 ID，新创建的线程可以通过调用 `pthread_self()` 获取其 ID。

原理说明

建议的 `pthread_create()` 替代方案是定义两个独立的操作：创建和启动。一些应用程序会发现这种行为更自然。特别是 Ada 语言，将任务的“创建”与其“激活”分开。

标准开发者出于多种原因拒绝了将操作分开：

- 启动线程所需的调用次数会从一个增加到两个，从而给不需要额外同步的应用程序带来额外负担。但是，通过启动状态属性的额外复杂性，可以避免第二次调用。
- 会引入一个额外的状态："已创建但未启动"。这将要求标准指定目标尚未开始执行时线程操作的行为。
- 对于需要这种行为的应用程序，可以使用当前提供的设施来模拟两个独立的步骤。`start_routine()` 可以通过等待由启动操作发信号的条件变量来同步。

Ada 实现者可以选择在 Ada 程序的两个点创建线程：创建任务对象时，或激活任务时（通常在 "begin" 处）。如果采用第一种方法，`start_routine()` 需要等待条件变量以接收开始"激活"的命令。第二种方法不需要这样的条件变量或额外同步。无论采用哪种方法，在创建任务对象时都需要创建单独的 Ada 任务控制块来保存会合队列等。

前面模型的扩展将允许在线程创建和启动之间修改线程状态。这将允许消除线程属性对象。这被拒绝是因为：

- 线程属性对象中的所有状态都必须能够为线程设置。这将需要定义修改线程属性的函数。设置线程所需的函数调用次数不会减少。实际上，对于使用相同属性创建所有线程的应用程序，设置线程所需的函数调用次数会大幅增加。使用线程属性对象允许应用程序进行一组属性设置函数调用。否则，每次线程创建都需要进行属性设置函数调用。
- 根据实现架构，设置线程状态的函数可能需要内核调用，或者出于其他实现原因无法实现为宏，从而增加线程创建的成本。
- 应用程序按类别隔离线程的能力将丢失。

另一个建议的替代方案使用类似于进程创建的模型，如"线程 fork"。fork 语义将提供更多灵活性，"create"函数可以通过执行线程 fork 然后立即调用线程所需的"启动例程"来实现。这种替代方案有以下问题：

- 对于许多实现，需要复制调用线程的整个栈，因为在许多架构中无法确定调用帧的大小。
- 效率降低，因为必须复制至少部分栈，尽管在大多数情况下线程永远不需要复制的上下文，因为它只是调用所需的启动例程。

如果实现检测到 `pthread_create()` 的 `attr` 参数指定的值不是指向已初始化的线程属性对象，建议函数应失败并报告 `[EINVAL]` 错误。

未来方向

无。

另请参阅

- `fork()`
- `pthread_exit()`
- `pthread_join()`

XBD 4.15.2 内存同步, `<pthread.h>`

变更历史

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_create()` 函数被标记为线程选项的一部分。

以下对 POSIX 实现的新要求来源于与单一 UNIX 规范的对齐：

- 添加了 `[EPERM]` 强制错误条件。

为了与 IEEE Std 1003.1d-1999 对齐，添加了线程 CPU 时间时钟语义。

为了与 ISO/IEC 9899:1999 标准对齐，在 `pthread_create()` 原型中添加了 `restrict` 关键字。

更新描述以明确浮点环境从创建线程继承。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/44，添加备用栈不被继承的文本。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/93，更新错误部分以删除强制 `[EINVAL]` 错误 (" `attr` 指定的值无效")，并添加可选 `[EINVAL]` 错误 (" `attr` 指定的属性无效")。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/94，添加应用程序使用部分。

Issue 7

`pthread_create()` 函数从线程选项移至基础。

删除了未初始化线程属性对象的 `[EINVAL]` 错误；此条件导致未定义行为。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0458 [302]。

应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0274 [849]。

1.140. pthread_detach — 分离线程

概要

```
#include <pthread.h>

int pthread_detach(pthread_t thread);
```

描述

`pthread_detach()` 函数应将线程 `thread` 从可结合状态 (joinable) 改变为分离状态 (detached)，向实现表明当线程终止时可以回收该线程的存储空间。如果 `thread` 尚未终止，`pthread_detach()` 不会导致其终止，但会防止该线程在终止时成为僵尸线程。

如果 `pthread_detach()` 的 `thread` 参数指定的值不引用可结合线程，则行为未定义。

返回值

如果调用成功，`pthread_detach()` 应返回 0；否则，应返回错误码以指示错误。

错误

`pthread_detach()` 函数不应返回 `[EINTR]` 错误码。

以下章节为参考信息。

示例

无。

应用程序用法

无。

基本原理

对于创建的每个线程，最终应调用 `pthread_join()` 或 `pthread_detach()` 函数，以便回收与线程相关的存储空间。

有人认为"分离"函数不是必需的；`detachstate` 线程创建属性就足够了，因为线程永远不需要动态分离。然而，至少在两种情况下会出现需求：

1. 在 `pthread_join()` 的取消处理程序中，几乎必须有一个 `pthread_detach()` 函数来分离 `pthread_join()` 正在等待的线程。没有它，处理程序将需要执行另一个 `pthread_join()` 来尝试分离线程，这既会无限期延迟取消处理，又会引入对 `pthread_join()` 的新调用，而这个新调用本身可能需要取消处理程序。在这种情况下，动态分离几乎是必需的。
2. 为了分离"初始线程"（在设置服务器线程的进程中可能需要这样做）。

如果实现检测到 `pthread_detach()` 的 `thread` 参数指定的值不引用可结合线程，建议函数应失败并报告 `[EINVAL]` 错误。

如果实现检测到在线程生命周期结束后使用线程 ID，建议函数应失败并报告 `[ESRCH]` 错误。

未来方向

无。

参见

- `pthread_join()`
- XBD `<pthread.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_detach()` 函数被标记为线程选项的一部分。

应用 IEEE Std 1003.1-2001/Cor 2-2004 项目 XSH/TC2/D6/95，更新错误部分，使 **[EINVAL]** 和 **[ESRCH]** 错误情况成为可选。

Issue 7

`pthread_detach()` 函数从线程选项移至基础部分。

应用 Austin Group Interpretation 1003.1-2001 #142，移除 **[ESRCH]** 错误条件。

移除不可结合线程的 **[EINVAL]** 错误；此条件导致未定义行为。

Issue 8

应用 Austin Group Defect 792，阐明分离活动线程可防止其在终止时成为僵尸线程。

应用 Austin Group Defect 1167，阐明在为线程调用 `pthread_detach()` 后，该线程不再可结合。

1.141. pthread_equal — 比较线程 ID

概要

```
#include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

描述

此函数用于比较线程 ID `t1` 和 `t2`。

返回值

如果 `t1` 和 `t2` 相等，`pthread_equal()` 函数应返回非零值；否则，应返回零。

如果 `t1` 或 `t2` 不是有效的线程 ID 且不等于 `PTHREAD_NULL`，则行为未定义。

错误

未定义任何错误。

`pthread_equal()` 函数不应返回 [EINTR] 错误码。

以下章节为参考信息。

示例

无。

应用程序使用

无。

基本原理

实现可以选择将线程 ID 定义为结构体。这比使用 `int` 类型提供了额外的灵活性和健壮性。例如，线程 ID 可以包含序列号，允许检测"悬空 ID"（已分离线程 ID 的副本）。由于 C 语言不支持结构体类型的比较，因此提供了 `pthread_equal()` 函数来比较线程 ID。

未来方向

无。

参见

- `pthread_create()`
- `pthread_self()`
- XBD `<pthread.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_equal()` 函数被标记为 Threads 选项的一部分。

Issue 7

- `pthread_equal()` 函数从 Threads 选项移动到 Base。

Issue 8

- 应用了 Austin Group Defect 599，更改了 RETURN VALUE 章节以提及 `PTHREAD_NULL`。

1.142. `pthread_exit` — 线程终止

概要

```
#include <pthread.h>

_Noreturn void pthread_exit(void *value_ptr);
```

描述

`pthread_exit()` 函数应当终止调用线程，并使 `value_ptr` 的值对任何与终止线程的成功连接操作可用。任何已推送但尚未弹出的取消清理处理程序应当按其被推送的相反顺序弹出并执行。在所有取消清理处理程序执行完毕后，如果线程有任何线程特定的数据（无论是与 `tss_t` 键类型还是 `pthread_key_t` 键类型相关联），应调用适当的析构函数，调用顺序未指定。线程终止不会释放任何应用程序可见的进程资源，包括但不限于互斥锁和文件描述符，也不会执行任何进程级别的清理操作，包括但不限于调用可能存在的任何 `atexit()` 例程。

当一个不是使用 `thrd_create()` 创建、也不是 `main()` 首次调用的线程从用于创建它的启动例程返回时，会隐式调用 `pthread_exit()`。函数的返回值将作为线程的退出状态。

如果从作为显式或隐式调用 `pthread_exit()` 结果而被调用的取消清理处理程序或析构函数中调用 `pthread_exit()`，其行为是未定义的。

线程终止后，访问线程的局部（自动）变量的结果是未定义的。因此，不应将退出线程的局部变量的引用用作 `pthread_exit()` 的 `value_ptr` 参数值。

在最后一个线程终止后，进程应以退出状态 0 退出。其行为应如同实现在线程终止时调用参数为零的 `_exit()`。

返回值

`pthread_exit()` 函数不能返回到其调用者。

错误

未定义任何错误。

示例

无。

应用程序用法

不应从使用 `thrd_create()` 创建的线程中调用 `pthread_exit()`，因为它们的退出状态具有不同的类型 (`int` 而不是 `void *`)。如果从初始线程中调用 `pthread_exit()` 并且它不是最后一个终止的线程，其他线程不应尝试使用 `thrd_join()` 获取其退出状态。

基本原理

使用 `pthread_create()` 启动的线程终止的正常机制是从启动它的 `pthread_create()` 调用中指定的例程返回。`pthread_exit()` 函数提供了线程在不要求从其启动例程返回的情况下终止的能力，从而提供了一个类似于 `_exit()` 的函数。

无论线程终止的方法如何，任何已推送但尚未弹出的取消清理处理程序都会被执行，并且任何存在的线程特定数据的析构函数也会被执行。POSIX.1-2024 要求取消清理处理程序按顺序弹出和调用。在所有取消清理处理程序执行完毕后，对于线程中存在的每个线程特定数据项，会以未指定的顺序调用线程特定数据析构函数。这种排序是必要的，因为取消清理处理程序可能依赖于线程特定数据。

由于状态的意义由应用程序确定（除非线程已被取消，此时为 `PTHREAD_CANCELED`），实现不知道什么是非法的状态值，这就是为什么不进行地址错误检查的原因。

未来方向

无。

参见

- `_exit()`
- `pthread_create()`
- `pthread_join()`
- `pthread_key_create()`
- `thrd_create()`
- `thrd_exit()`
- `tss_create()`
- `<pthread.h>`

更改历史

首次在 Issue 5 中发布。包含用于与 POSIX 线程扩展对齐。

Issue 6

`pthread_exit()` 函数被标记为 Threads 选项的一部分。

Issue 7

`pthread_exit()` 函数从 Threads 选项移动到 Base。

Issue 8

应用 Austin Group 缺陷 1302，在概要中添加了 `_Noreturn`，并更新页面以考虑 `<threads.h>` 接口的添加。

1.143. pthread_getconcurrency

概要

```
#include <pthread.h>

int pthread_getconcurrency(void);
```

描述

进程中的未绑定线程可能需要也可能不需要同时激活。默认情况下，线程实现确保有足够的线程处于激活状态，以便进程能够继续执行。虽然这样做可以节省系统资源，但可能不会产生最有效的并发级别。

`pthread_setconcurrency()` 函数允许应用程序将其期望的并发级别 `new_level` 通知给线程实现。由于此函数调用，实现提供的实际并发级别是未指定的。

如果 `new_level` 为零，则会导致实现自行维护并发级别，就像从未调用过 `pthread_setconcurrency()` 一样。

`pthread_getconcurrency()` 函数应返回先前调用 `pthread_setconcurrency()` 函数所设置的值。如果先前没有调用过 `pthread_setconcurrency()` 函数，此函数应返回零，表示实现正在维护并发级别。

调用 `pthread_setconcurrency()` 应通知实现其期望的并发级别。实现应将此作为提示，而不是要求。

返回值

`pthread_getconcurrency()` 函数应始终返回先前调用 `pthread_setconcurrency()` 所设置的并发级别。如果从未调用过 `pthread_setconcurrency()` 函数，`pthread_getconcurrency()` 应返回零。

错误

`pthread_getconcurrency()` 未定义任何错误。

示例

示例 1：获取当前并发级别

```
#include <stdio.h>
#include <pthread.h>

int main(void) {
    int level;

    level = pthread_getconcurrency();
    printf("Current concurrency level: %d\n", level);

    if (level == 0) {
        printf("Implementation is maintaining concurrency level
    }

    return 0;
}
```

示例 2：设置和获取并发级别

```
#include <stdio.h>
#include <pthread.h>

int main(void) {
    int ret, level;

    // Set desired concurrency level
    ret = pthread_setconcurrency(4);
    if (ret != 0) {
        printf("Failed to set concurrency level\n");
        return 1;
    }

    // Get current concurrency level
    level = pthread_getconcurrency();
    printf("Concurrency level set to: %d\n", level);
```

```
// Reset to implementation-discretion
ret = pthread_setconcurrency(0);
if (ret == 0) {
    level = pthread_getconcurrency();
    printf("After reset, concurrency level: %d\n", level);
}

return 0;
}
```

应用程序使用

应用程序应使用 `pthread_getconcurrency()` 来确定先前使用 `pthread_setconcurrency()` 设置的当前并发级别。该函数特别适用于：

- 检查是否设置了特定的并发级别
- 验证先前 `pthread_setconcurrency()` 调用的效果
- 调试和监视线程并发行为

基本原理

如果实现不支持在多个内核调度实体之上多路复用用户线程，则提供 `pthread_setconcurrency()` 和 `pthread_getconcurrency()` 函数是为了源代码兼容性，但在调用时它们应不起作用。为了保持函数语义，当调用 `pthread_setconcurrency()` 时保存 `new_level` 参数，以便后续调用 `pthread_getconcurrency()` 应返回相同的值。

未来方向

无。

另请参见

- `pthread_setconcurrency()`
- `pthread_create()`
- `pthread_attr_init()`

版权

本文部分内容摘录并转载自 IEEE Std 1003.1, 2013 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, Copyright (C) 2013 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group. 如果此版本与原始 IEEE 和 The Open Group 标准之间存在任何差异，应以原始 IEEE 和 The Open Group 标准为准。原始标准可在线获取：<http://www.opengroup.org/unix/online.html>。

本页面中出现的任何印刷或格式错误很可能是在将源文件转换为手册页格式时引入的。要报告此类错误，请参见 https://www.kernel.org/doc/man-pages/reporting_bugs.html。

1.144. pthread_getcpuclockid

SYNOPSIS (概要)

```
#include <pthread.h>
#include <time.h>

int pthread_getcpuclockid(pthread_t thread_id, clockid_t *clock
```



DESCRIPTION (描述)

如果由 `thread_id` 指定的线程存在, `pthread_getcpuclockid()` 函数应将该线程的 CPU 时间时钟的时钟 ID 返回到 `clock_id` 中。

RETURN VALUE (返回值)

成功完成后, `pthread_getcpuclockid()` 应返回零; 否则, 应返回错误号以指示错误。

ERRORS (错误)

未定义错误。

以下章节为参考信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

`pthread_getcpuclockid()` 函数是线程 CPU 时间时钟选项的一部分, 不需要在所有实现中提供。

RATIONALE (基本原理)

如果实现在线程生命周期结束后检测到线程 ID 的使用，建议该函数应失败并报告 [ESRCH] 错误。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `clock_getcpuclockid()`
- `clock_getres()`
- `timer_create()`

XBD `<pthread.h>` , `<time.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 6。源自 IEEE Std 1003.1d-1999。

在 SYNOPSIS 中，不再需要包含 `<sys/types.h>`。

Issue 7

由于线程选项现在是基础的一部分，`pthread_getcpuclockid()` 函数仅标记为线程 CPU 时间时钟选项的一部分。

应用 Austin Group Interpretation 1003.1-2001 #142，移除了 [ESRCH] 错误条件。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0275 [757]。

1.145. `pthread_getschedparam`, `pthread_setschedparam` — 动态线程调度参数访问 (REALTIME THREADS)

SYNOPSIS

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread,
                          int *restrict policy,
                          struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread,
                          int policy,
                          const struct sched_param *param);
```

DESCRIPTION

`pthread_getschedparam()` 和 `pthread_setschedparam()` 函数分别用于获取和设置多线程进程中单个线程的调度策略和参数。对于 SCHED_FIFO 和 SCHED_RR，`sched_param` 结构中唯一必需的成员是优先级 `sched_priority`。对于 SCHED_OTHER，受影响的调度参数由实现定义。

`pthread_getschedparam()` 函数应检索由 `thread` 给定线程 ID 的线程的调度策略和调度参数，并分别将这些值存储在 `policy` 和 `param` 中。从 `pthread_getschedparam()` 返回的优先级值应为影响目标线程的最近一次 `pthread_setschedparam()`、`pthread_setschedprio()` 或 `pthread_create()` 调用指定的值。它不应反映由于任何优先级继承或上限函数导致的对其优先级的任何临时调整。`pthread_setschedparam()` 函数应将由 `thread` 给定线程 ID 的线程的调度策略和关联调度参数设置为 `policy` 和 `param` 中分别提供的策略和关联参数。

`policy` 参数可以具有 SCHED_OTHER、SCHED_FIFO 或 SCHED_RR 的值。SCHED_OTHER 策略的调度参数由实现定义。SCHED_FIFO 和 SCHED_RR 策略应具有单一调度参数 `priority`。

如果定义了 `_POSIX_THREAD_SPORADIC_SERVER`，那么 `policy` 参数可以具有 SCHED_SPORADIC 的值，但 `pthread_setschedparam()` 函数的例外是，如果在调用时调度策略不是 SCHED_SPORADIC，则是否支持该函数由实现定

义；换句话说，实现不需要允许应用程序动态将调度策略更改为 SCHED_SPORADIC。偶发服务器调度策略具有关联参数 `sched_ss_low_priority`、`sched_ss_repl_period`、`sched_ss_init_budget`、`sched_priority` 和 `sched_ss_max_repl`。指定的 `sched_ss_repl_period` 必须大于或等于指定的 `sched_ss_init_budget` 函数才能成功；如果不是，则函数应失败。`sched_ss_max_repl` 的值必须在 [1,{SS_REPL_MAX}] 的包含范围内函数才能成功；如果不是，函数应失败。`sched_ss_repl_period` 和 `sched_ss_init_budget` 值是否按此函数提供的值存储或舍入以与正在使用的时钟分辨率对齐是未指定的。

如果 `pthread_setschedparam()` 函数失败，则目标线程的调度参数不应更改。

RETURN VALUE

如果成功，`pthread_getschedparam()` 和 `pthread_setschedparam()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

`pthread_setschedparam()` 函数应在以下情况下失败：

- **ENOTSUP**: 尝试将策略或调度参数设置为不支持的值。
- **ENOTSUP**: 尝试将调度策略动态更改为 SCHED_SPORADIC，而实现不支持此更改。

`pthread_setschedparam()` 函数可能在以下情况下失败：

- **EINVAL**: 由 `policy` 指定的值或与调度策略 `policy` 关联的某个调度参数无效。
- **EPERM**: 调用者没有适当的权限来设置指定线程的调度参数或调度策略。
- **EPERM**: 实现不允许应用程序将某个参数修改为指定的值。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

如果实现在线程生命周期结束后检测到线程 ID 的使用，建议函数失败并报告 [ESRCH] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_setschedprio()`
- `sched_getparam()`
- `sched_getscheduler()`
- XBD `<pthread.h>`
- XBD `<sched.h>`

CHANGE HISTORY

首次发布于 Issue 5

为与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_getschedparam()` 和 `pthread_setschedparam()` 函数被标记为线程和线程执行调度选项的一部分。
- [ENOSYS] 错误条件已被移除，因为如果实现不支持线程执行调度选项，则无需提供存根。
- 应用了 Open Group Corrigendum U026/2，更正了 `pthread_setschedparam()` 函数的原型，使其第二个参数为 `int` 类

型。

- 为与 IEEE Std 1003.1d-1999 对齐而添加了 SCHED_SPORADIC 调度策略。
- 为与 ISO/IEC 9899:1999 标准对齐而向 `pthread_getschedparam()` 原型添加了 `restrict` 关键字。
- 应用了 Open Group Corrigendum U047/1。
- 应用了 IEEE PASC Interpretation 1003.1 #96，注意优先级值也可以通过调用 `pthread_setschedprio()` 函数来设置。

Issue 7

- `pthread_getschedparam()` 和 `pthread_setschedparam()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。
 - 应用了 Austin Group Interpretation 1003.1-2001 #119，阐明了 `sched_ss_repl_period` 和 `sched_ss_init_budget` 值的准确性要求。
 - 应用了 Austin Group Interpretation 1003.1-2001 #142，移除了 [ESRCH] 错误条件。
 - 应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0459 [314]。
 - 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0276 [757]。
-

1.146. `pthread_getspecific`, `pthread_setspecific` — 线程特定数据管理

概要

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);
int pthread_setspecific(pthread_key_t key, const void *value);
```

描述

`pthread_getspecific()` 函数应返回当前绑定到指定 `key` 的值，该操作代表调用线程执行。

`pthread_setspecific()` 函数应将线程特定的 `value` 与通过先前调用 `pthread_key_create()` 获得的 `key` 关联。不同的线程可能将不同的值绑定到同一个键。这些值通常是指向动态分配内存块的指针，这些内存块已保留供调用线程使用。

使用不是从 `pthread_key_create()` 获得的 `key` 值调用 `pthread_getspecific()` 或 `pthread_setspecific()`，或在 `key` 已被 `pthread_key_delete()` 删除后调用这些函数，其效果是未定义的。

`pthread_getspecific()` 和 `pthread_setspecific()` 都可以从线程特定数据析构函数中调用。对正在销毁的线程特定数据键调用 `pthread_getspecific()` 应返回值 `NULL`，除非该值在析构函数启动后通过调用 `pthread_setspecific()` 被更改。从线程特定数据析构例程调用 `pthread_setspecific()` 可能导致存储丢失（在至少 `PTHREAD_DESTRUCTOR_ITERATIONS` 次析构尝试后）或无限循环。

两个函数都可以实现为宏。

返回值

`pthread_getspecific()` 函数应返回与给定 `key` 关联的线程特定数据值。如果没有线程特定数据值与 `key` 关联，则应返回值 `NULL`。

如果成功，`pthread_setspecific()` 函数应返回零；否则，应返回错误号以指示错误。

错误

`pthread_getspecific()` 不返回任何错误。

`pthread_setspecific()` 函数在以下情况下应失败：

- [ENOMEM]
- 没有足够的内存来将非 NULL 值与键关联。

`pthread_setspecific()` 函数不应回 [EINTR] 错误代码。

示例

无。

应用用法

无。

原理

`pthread_getspecific()` 的性能和易用性对于依赖在线程特定数据中维护状态的函数至关重要。由于它不需要检测任何错误，并且由于唯一可检测的错误是使用无效键，`pthread_getspecific()` 函数的设计优先考虑速度和简单性，而不是错误报告。

如果实现检测到 `pthread_setspecific()` 的 `key` 参数指定的值不是从 `pthread_key_create()` 获得的键值，或者引用了已被 `pthread_key_delete()` 删除的键，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

另请参见

- `pthread_key_create()`
- `<pthread.h>`

变更历史

首次发布于 Issue 5

包含是为了与 POSIX 线程扩展保持一致。

Issue 6

- `pthread_getspecific()` 和 `pthread_setspecific()` 函数被标记为线程选项的一部分。
- 应用了 IEEE PASC Interpretation 1003.1c #3 (Part 6)，更新了描述部分。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/96，更新了错误部分，使 [ENOMEM] 错误情况从"将值与键关联"更改为"将非 NULL 值与键关联"。

Issue 7

- 应用了 Austin Group Interpretation 1003.1-2001 #063，更新了错误部分。
- `pthread_getspecific()` 和 `pthread_setspecific()` 函数从线程选项移动到基础部分。
- 删除了对于不是从 `pthread_key_create()` 获得或已被 `pthread_key_delete()` 删除的键值的 [EINVAL] 错误；这种情况会导致未定义行为。

1.147. pthread_join — 等待线程终止

概要

```
#include <pthread.h>

int pthread_join(pthread_t thread, void **value_ptr);
```

描述

`pthread_join()` 函数应当挂起调用线程的执行，直到目标线程终止，除非目标线程已经终止。当成功调用 `pthread_join()` 且 `value_ptr` 参数不为 NULL 时，终止线程传递给 `pthread_exit()` 的值应当在 `value_ptr` 引用的位置中可用。当 `pthread_join()` 成功返回时，目标线程已被终止。同时指定相同目标线程的多次 `pthread_join()` 调用的结果是未定义的。如果调用 `pthread_join()` 的线程被取消，则目标线程不应被分离。

僵尸线程是否计入 {PTHREAD_THREADS_MAX} 是未指定的。

如果传递给 `pthread_join()` 的 `thread` 参数值不指向可连接的线程，则行为是未定义的。

如果传递给 `pthread_join()` 的 `thread` 参数值指向调用线程，则行为是未定义的。

如果 `thread` 指向使用 `thrd_create()` 创建的线程，且该线程通过从其启动例程返回而终止，或已经终止，则 `pthread_join()` 的行为是未定义的。如果 `thread` 指向通过调用 `thrd_exit()` 终止的线程，或已经终止的线程，则 `pthread_join()` 的行为是未定义的。

返回值

如果成功，`pthread_join()` 函数应当返回零；否则，应当返回错误编号以指示错误。

错误

`pthread_join()` 函数可能在以下情况下失败：

- [EDEADLK] 检测到死锁。

`pthread_join()` 函数不应返回 [EINTR] 错误码。

示例

线程创建和删除的示例如下：

```
typedef struct {
    int *ar;
    long n;
} subarray;

void *
incer(void *arg)
{
    long i;

    for (i = 0; i < ((subarray *)arg)->n; i++)
        ((subarray *)arg)->ar[i]++;
}

int main(void)
{
    int      ar[1000000];
    pthread_t th1, th2;
    subarray sb1, sb2;

    sb1.ar = &ar[0];
    sb1.n  = 500000;
    (void) pthread_create(&th1, NULL, incer, &sb1);

    sb2.ar = &ar[500000];
    sb2.n  = 500000;
    (void) pthread_create(&th2, NULL, incer, &sb2);

    (void) pthread_join(th1, NULL);
    (void) pthread_join(th2, NULL);
    return 0;
}
```

应用程序用法

无。

原理

`pthread_join()` 函数是一个便利功能，在多线程应用程序中已被证明是有用的。确实，如果没有提供此函数，程序员可以通过向 `start_routine()` 传递额外状态作为参数来模拟此函数。终止线程将设置一个标志来指示终止并广播作为该状态一部分的条件；连接线程将在该条件变量上等待。虽然这种技术允许线程等待更复杂的条件（例如，等待多个线程终止），但等待单个线程终止被认为是有广泛用途的。此外，包含 `pthread_join()` 函数绝不妨碍程序员编写此类复杂等待。因此，虽然不是原语，但在 POSIX.1-2024 中包含 `pthread_join()` 被认为是有价值的。

`pthread_join()` 函数提供了一个简单的机制，允许应用程序等待线程终止。线程终止后，应用程序可以选择清理线程使用的资源。例如，在 `pthread_join()` 返回后，可以回收任何应用程序提供的栈存储。

对于每个以 `detachstate` 属性设置为 `PTHREAD_CREATE_JOINABLE` 创建的线程，最终应调用 `pthread_join()` 或 `pthread_detach()` 函数，以便回收与线程关联的存储。

`pthread_join()` 和取消之间的交互是明确定义的，原因如下：

- `pthread_join()` 函数，与所有其他非异步取消安全函数一样，只能在延迟取消类型下调用。
- 在禁用取消状态下不能发生取消。

因此，只需要考虑默认的取消状态。如指定的那样，`pthread_join()` 调用要么被取消，要么成功，但不能两者都有。对于应用程序来说，区别是明显的，因为要么运行取消处理程序，要么 `pthread_join()` 返回。由于 `pthread_join()` 是在延迟取消状态下调用的，因此不存在竞争条件。

如果实现检测到传递给 `pthread_join()` 的 `thread` 参数值不指向可连接的线程，建议函数应失败并报告 `[EINVAL]` 错误。

如果实现检测到传递给 `pthread_join()` 的 `thread` 参数值指向调用线程，建议函数应失败并报告 `[EDEADLK]` 错误。

如果实现检测到在线程 ID 生命周期结束后使用该线程 ID，建议函数应失败并报告 `[ESRCH]` 错误。

`pthread_join()` 函数不能用于获取使用 `thrd_create()` 创建并通过从其启动例程返回而终止的线程的退出状态，也不能用于获取通过调用 `thrd_exit()` 终止的线程的退出状态，因为此类线程具有 `int` 退出状态，而不是 `pthread_join()` 通过其 `value_ptr` 参数返回的 `void *` 类型。

未来方向

无。

另请参阅

- `pthread_create()`
- `thrd_create()`
- `thrd_exit()`
- `wait()`
- XBD 4.15.2 内存同步, `<pthread.h>`

更改历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_join()` 函数被标记为线程选项的一部分。

应用 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/97, 更新 ERRORS 部分, 使 [EINVAL] 错误成为可选, 并从中删除"实现已检测到"字样。

Issue 7

`pthread_join()` 函数从线程选项移动到基础。

应用 Austin Group 解释 1003.1-2001 #142, 删除 [ESRCH] 错误条件。

删除非可连接线程的 [EINVAL] 错误; 此条件导致未定义行为。

删除调用线程的 [EDEADLK] 错误; 此条件导致未定义行为。

Issue 8

应用 Austin Group 缺陷 792, 将"已退出但未连接的线程"更改为"僵尸线程"。

应用 Austin Group 缺陷 1302, 更新页面以考虑 `<threads.h>` 接口的添加。

1.148. pthread_key_create — 线程特定数据键创建

概要 (SYNOPSIS)

```
#include <pthread.h>

int pthread_key_create(pthread_key_t *key, void (*destructor)(v
```

描述 (DESCRIPTION)

`pthread_key_create()` 函数应创建一个对进程中所有线程都可见的线程特定数据键。 `pthread_key_create()` 提供的键值是不透明对象，用于定位线程特定数据。 虽然不同的线程可能使用相同的键值，但由于 `pthread_setspecific()` 绑定到键的值是按线程维护的，并在调用线程的生命周期内持续存在。

创建键时，在所有活动线程中，新键应与 NULL 值关联。创建线程时，在新线程中，所有已定义的键应与 NULL 值关联。

可以为每个键值关联一个可选的析构函数。线程退出时，如果键值具有非 NULL 的析构函数指针，且该线程具有与该键关联的非 NULL 值，则该键的值被设置为 NULL，然后调用指向的函数，将先前关联的值作为其唯一参数。当线程退出时如果存在多个析构函数，则析构函数调用的顺序未指定。

如果，在为所有具有关联析构函数的非 NULL 值调用完所有析构函数后，仍然存在一些具有关联析构函数的非 NULL 值，则重复该过程。如果，在对未处理的非 NULL 值进行了至少 {PTHREAD_DESTRUCTOR_ITERATIONS} 次析构函数调用后，仍然存在一些具有关联析构函数的非 NULL 值，实现可以停止调用析构函数，或者可以继续调用析构函数，直到不存在具有关联析构函数的非 NULL 值为止，即使这可能导致无限循环。

返回值 (RETURN VALUE)

如果成功，`pthread_key_create()` 函数应将新创建的键值存储在 *key 处并返回零。否则，应返回错误号以指示错误。

错误 (ERRORS)

`pthread_key_create()` 函数在以下情况下可能失败：

- **[EAGAIN]**
- 系统缺乏创建另一个线程特定数据键所需的必要资源，或系统强加的每个进程键总数限制 {PTHREAD_KEYS_MAX} 已被超过。
- **[ENOMEM]**
- 内存不足，无法创建该键。

`pthread_key_create()` 函数不应返回 [EINTR] 错误码。

示例 (EXAMPLES)

以下示例演示了一个函数，在首次调用时初始化线程特定数据键，并将线程特定对象与每个调用线程关联，必要时初始化此对象。

```
static pthread_key_t key;
static pthread_once_t key_once = PTHREAD_ONCE_INIT;

static void
make_key()
{
    (void) pthread_key_create(&key, NULL);
}

func()
{
    void *ptr;

    (void) pthread_once(&key_once, make_key);
    if ((ptr = pthread_getspecific(key)) == NULL) {
        ptr = malloc(OBJECT_SIZE);
        ...
        (void) pthread_setspecific(key, ptr);
    }
    ...
}
```

注意，键必须在使用 `pthread_getspecific()` 或 `pthread_setspecific()` 之前进行初始化。`pthread_key_create()` 调用可以显式地在模块初始化例程中进行，也可以像本例那样在模块的第一次调用

时隐式进行。在键初始化之前使用键的任何尝试都是编程错误，使以下代码不正确。

```
static pthread_key_t key;

func()
{
    void *ptr;

    /* 键未初始化！！！ 这不会工作！！！ */
    if ((ptr = pthread_getspecific(key)) == NULL &&
        pthread_setspecific(key, NULL) != 0) {
        pthread_key_create(&key, NULL);
        ...
    }
}
```

应用程序用法 (APPLICATION USAGE)

无。

基本原理 (RATIONALE)

析构函数

通常，为表示特定线程绑定到键的值是指向为调用线程动态分配的存储的指针。与 `pthread_key_create()` 一起指定的析构函数旨在在线程退出时释放此存储。线程取消清理处理程序不能用于此目的，因为线程特定数据可能在取消清理处理程序操作的词法作用域之外持续存在。

如果在线程生命周期期间需要更新与键关联的值，可能需要在新值绑定之前释放与旧值关联的存储。虽然 `pthread_setspecific()` 函数可以自动执行此操作，但此功能不够常用，不足以证明增加的复杂性是合理的。相反，程序员负责释放过期的存储：

```
old = pthread_getspecific(key);
new = allocate();
destructor(old);
pthread_setspecific(key, new);
```

注意：

如果在启用异步取消的情况下运行，上述示例可能会泄漏存储。如果在获取和设置之间没有取消点，则在默认取消状态下不会发生此类问题。

没有析构函数安全函数的概念。如果应用程序不从信号处理程序调用 `pthread_exit()`，或者在调用异步不安全函数时阻止任何可能调用 `pthread_exit()` 的信号处理程序的信号，则可以从析构函数安全地调用所有函数。

非幂等数据键创建

曾有请求要求使 `pthread_key_create()` 对于给定的键地址参数是幂等的。这将允许应用程序为给定的键地址多次调用 `pthread_key_create()`，并保证只创建一个键。这样做需要键值之前被初始化（可能在编译时）为已知的空值，并且需要基于键参数的地址和内容执行隐式互斥，以保证只创建一个键。

不幸的是，隐式互斥不仅限于 `pthread_key_create()`。在许多实现上，`pthread_getspecific()` 和 `pthread_setspecific()` 也必须执行隐式互斥，以防止使用未完全存储或尚未可见的键值。这可能会显著增加重要操作的成本，特别是 `pthread_getspecific()`。

因此，这个提案被拒绝了。`pthread_key_create()` 函数不执行隐式同步。程序员有责任确保在使用键之前每个键只调用一次该函数。已经有几种直接的机制可用于完成此操作，包括调用显式模块初始化函数、使用互斥锁和使用 `pthread_once()`。这不会给程序员带来重大负担，不会引入可能令人困惑的 *ad hoc* 隐式同步机制，并且可能允许更常用的线程特定数据操作更高效。

未来方向 (FUTURE DIRECTIONS)

无。

另请参阅 (SEE ALSO)

- `pthread_getspecific()`
- `pthread_key_delete()`
- `<pthread.h>`

变更历史 (CHANGE HISTORY)

在第 5 版中首次发布

包含以与 POSIX 线程扩展对齐。

第 6 版

`pthread_key_create()` 函数被标记为线程选项的一部分。

应用了 IEEE PASC Interpretation 1003.1c #8，更新了描述部分。

第 7 版

`pthread_key_create()` 函数从线程选项移动到基础。

第 8 版

应用了 Austin Group Defect 1059，更改了基本原理部分的示例代码。

1.149. pthread_key_delete

概要

```
#include <pthread.h>

int pthread_key_delete(pthread_key_t key);
```

描述

`pthread_key_delete()` 函数应删除一个之前由 `pthread_key_create()` 返回的线程特定数据键。与 `key` 关联的线程特定数据值在调用 `pthread_key_delete()` 时不必为 NULL。释放任何应用程序存储或执行与已删除键或任何线程中关联的线程特定数据相关的数据结构的清理操作是应用程序的责任；此清理可以在调用 `pthread_key_delete()` 之前或之后进行。在调用 `pthread_key_delete()` 之后任何使用 `key` 的尝试都会导致未定义行为。

`pthread_key_delete()` 函数应可从析构函数内部调用。
`pthread_key_delete()` 不会调用任何析构函数。任何可能与 `key` 关联的析构函数在线程退出时都不再被调用。

返回值

成功完成后，`pthread_key_delete()` 应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_key_delete()` 函数可能在以下情况下失败：

- **EINVAL** - `key` 指定的值不是从 `pthread_key_create()` 获得的键值，或引用了已被 `pthread_key_delete()` 删除的键。

示例

未提供示例。

应用程序用法

包含线程特定数据键删除功能是为了允许释放与未使用的线程特定数据键关联的资源。未使用的线程特定数据键可能在多种情况下出现，例如当分配了键的动态加载模块被卸载时。

符合标准的应用程序负责执行与要删除的键关联的数据结构所需的任何清理操作，包括线程特定数据值引用的数据。`pthread_key_delete()` 不执行此类清理。特别地，不调用析构函数。这种职责划分有几个原因：

1. 用于在线程退出时释放线程特定数据的关联析构函数只有在分配线程特定数据的线程中调用时才能保证正确工作。（析构函数本身可能使用线程特定数据。）因此，它们不能用于在键删除时释放其他线程中的线程特定数据。试图在键删除时让其他线程调用它们需要异步中断其他线程。但由于被中断的线程可能处于任意状态，包括持有析构函数运行所需的锁，这种方法会失败。一般来说，没有安全的机制可以让实现在键删除时释放线程特定数据。
2. 即使有安全释放与要删除的键关联的线程特定数据的方法，这样做也需要实现能够枚举具有非 NULL 数据的线程，并可能阻止在键删除过程中创建更多线程特定数据。这种特殊情况会在正常情况下导致额外的同步，而这是不必要的。

应用程序要知道删除键是安全的，它必须知道所有可能曾经使用该键的线程都不会再次尝试使用它。例如，如果所有客户端线程都调用了清理过程，声明它们完成了正在关闭的模块的工作（可能通过将引用计数设置为零），它就可以知道这一点。

基本原理

如果实现检测到传递给 `pthread_key_delete()` 的 `key` 参数值不是从 `pthread_key_create()` 获得的键值，或引用了已被 `pthread_key_delete()` 删除的键，建议函数应失败并报告 `[EINVAL]` 错误。

未来方向

无。

另请参阅

[pthread_key_create\(\)](#)

XSI

无。

版权

本文部分内容是从 IEEE Std 1003.1-2017 for POSIX.1-2017, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 2018 Edition 中以电子形式转载和复制的，版权归属于电气和电子工程师学会有限公司和 The Open Group (C) 2018。如果此版本与原始 IEEE 和 The Open Group 标准之间存在任何差异，应以原始 IEEE 和 The Open Group 标准为准。原始标准可在线获取：<http://www.opengroup.org/unix/online.html>。

本页上出现的任何印刷或格式错误很可能是在将源文件转换为手册页格式时引入的。要报告此类错误，请参见 https://www.kernel.org/doc/man-pages/reporting_bugs.html。

1.150. pthread_kill

概要

```
#include <signal.h>

int pthread_kill(pthread_t thread, int sig);
```

描述

`pthread_kill()` 函数应请求将信号传递给指定的线程。如果 `thread` 是僵尸线程，这不应被视为错误。

返回值

成功完成后，函数应返回零值。否则，函数应返回错误编号。如果 `pthread_kill()` 函数失败，则不会发送任何信号。

错误

`pthread_kill()` 函数可能在以下情况下失败：

- **[EINVAL]**
- `sig` 参数的值为零。

`pthread_kill()` 函数应在以下情况下失败：

- **[EINVAL]**
- `sig` 参数的值非零，并且是无效或不支持的信号编号。

`pthread_kill()` 函数不应返回 **[EINTR]** 错误代码。

以下章节为参考信息。

示例

无。

应用程序用法

`pthread_kill()` 函数提供了一种机制，用于异步地将信号定向到调用进程中的线程。例如，这可以由一个线程用来影响向一组线程广播信号传递。

请注意，`pthread_kill()` 仅导致信号在给定线程的上下文中被处理；信号动作（终止或停止）影响整个进程。

原理

如果实现在线程生命周期结束后检测到使用线程 ID，建议函数应失败并报告 **[ESRCH]** 错误。

历史实现在使用表示僵尸线程的线程 ID 调用 `pthread_kill()` 时的结果各不相同。一些实现在此类调用中表示成功，而其他实给出 **[ESRCH]** 错误。由于 POSIX.1-2024 本卷中线程生命周期的定义涵盖了僵尸线程，因此描述的 **[ESRCH]** 错误在这种情况下是不合适的，给出此错误的实现不符合规范。特别是，这意味着应用程序不能通过使用零 `sig` 参数调用 `pthread_kill()` 来让一个线程检查另一个线程的终止，而当 `sig` 为零时，实现可以通过返回 **[EINVAL]** 来表明这是不可能的。

未来方向

无。

另请参阅

- `kill()`
- `pthread_self()`
- `raise()`

XBD `<signal.h>`

变更历史

首次发布于第 5 版。为与 POSIX 线程扩展对齐而包含在内。

第 6 版

`pthread_kill()` 函数被标记为线程选项的一部分。

添加了应用程序用法章节。

第 7 版

`pthread_kill()` 函数从线程选项移动到基础部分。

应用了 Austin Group 解释 1003.1-2001 #142，移除了 **[ESRCH]** 错误条件。

应用了 POSIX.1-2008，技术勘误表 2，XSH/TC2-2008/0277 [765]。

第 8 版

应用了 Austin Group 缺陷 792，添加了将僵尸线程的线程 ID 传递给 `pthread_kill()` 不视为错误的要求。

应用了 Austin Group 缺陷 1214，允许当 `sig` 参数为零时 `pthread_kill()` 失败并返回 **[EINVAL]**。

1.151. pthread_mutex_destroy

概要

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr)

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

描述

`pthread_mutex_destroy()` 函数应销毁由 `mutex` 引用的互斥锁对象；互斥锁对象实际上变为未初始化状态。实现可能使 `pthread_mutex_destroy()` 将由 `mutex` 引用的对象设置为无效值。

已销毁的互斥锁对象可以使用 `pthread_mutex_init()` 重新初始化；在销毁后以其他方式引用该对象的结果是未定义的。

销毁一个已初始化且未锁定的互斥锁是安全的。尝试销毁一个已锁定的互斥锁，或者另一个线程正尝试锁定的互斥锁，或者另一个线程正在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用的互斥锁，会导致未定义行为。

`pthread_mutex_init()` 函数应使用由 `attr` 指定的属性初始化由 `mutex` 引用的互斥锁。如果 `attr` 为 NULL，则使用默认的互斥锁属性；效果应与传递默认互斥锁属性对象的地址相同。成功初始化后，互斥锁的状态变为已初始化且未锁定。

further requirements。

尝试初始化一个已经初始化的互斥锁会导致未定义行为。

在默认互斥锁属性合适的情况下，可以使用宏 `PTHREAD_MUTEX_INITIALIZER` 来初始化互斥锁。效果应等同于通过调用参数 `attr` 指定为 NULL 的 `pthread_mutex_init()` 进行动态初始化，但不执行错误检查。

如果传递给 `pthread_mutex_destroy()` 的 `mutex` 参数值不引用已初始化的互斥锁，则行为未定义。

如果传递给 `pthread_mutex_init()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，则行为未定义。

返回值

如果成功，`pthread_mutex_destroy()` 和 `pthread_mutex_init()` 函数应返回零；否则，应返回错误号以指示错误。

错误

`pthread_mutex_destroy()` 函数可能在以下情况失败：

- **EBUSY** - 实现检测到尝试销毁由 `mutex` 引用的对象，而该对象已被锁定或被引用（例如，正在被另一个线程在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用）。
- **EINVAL** - 由 `mutex` 指定的值不引用已初始化的互斥锁对象。

`pthread_mutex_init()` 函数可能在以下情况失败：

- **EBUSY** - 实现检测到尝试重新初始化由 `mutex` 引用的对象，该对象是之前已初始化但尚未销毁的互斥锁。
- **EINVAL** - 由 `attr` 指定的值不引用已初始化的互斥锁属性对象，或者由 `mutex` 指定的值不引用已初始化的互斥锁。
- **ENOMEM** - 内存不足，无法初始化互斥锁。

这些函数不应返回错误代码 [EINTR]。

示例

静态初始化

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;  
  
void foo()  
{  
    pthread_mutex_lock(&foo_mutex);  
    /* 执行工作。 */
```

```
    pthread_mutex_unlock(&foo_mutex);  
}
```

动态初始化

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;  
static pthread_mutex_t foo_mutex;  
  
void foo_init()  
{  
    pthread_mutex_init(&foo_mutex, NULL);  
}  
  
void foo()  
{  
    pthread_once(&foo_once, foo_init);  
    pthread_mutex_lock(&foo_mutex);  
    /* 执行工作。 */  
    pthread_mutex_unlock(&foo_mutex);  
}
```

应用程序用法

如果传递给 `pthread_mutex_destroy()` 的 `mutex` 参数值不引用已初始化的互斥锁，则行为未定义。

如果传递给 `pthread_mutex_init()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，则行为未定义。

基本原理

可能的替代实现

POSIX.1-2024 支持几种互斥锁的替代实现。实现可以直接将锁存储在 `pthread_mutex_t` 类型的对象中。或者，实现可以在堆中存储锁，而仅在互斥锁对象中存储指针、句柄或唯一 ID。每种实现都有其优势，或者在特定的硬件配置下可能是必需的。为了编写不受此选择影响的可移植代码，POSIX.1-2024 不为此类型定义赋值或相等性，并使用术语“初始化”来加强（更具限制性的）概念，即锁实际上可能驻留在互斥锁对象本身中。

请注意，这避免了对互斥锁或条件变量类型的过度规范，并促使该类型的不透明性。

允许但不要求实现使 `pthread_mutex_destroy()` 将非法值存储到互斥锁中。这可能有助于检测尝试锁定（或以其他方式引用）已销毁互斥锁的错误程序。

支持错误检查和性能之间的权衡

许多可能发生的错误情况不需要被实现检测到，以便让实现根据其特定应用程序和执行环境的需求在性能和错误检查程度之间进行权衡。作为一般规则，要求检测由系统引起的条件（如内存不足），但由错误编码的应用程序引起的条件（如未能提供足够的同步以防止互斥锁在使用时被删除）被指定为导致未定义行为。

互斥锁和条件变量的静态初始化符

为静态分配的同步对象提供静态初始化允许具有私有静态同步变量的模块避免运行时初始化测试和开销。此外，它简化了自初始化模块的编码。此类模块在 C 库中很常见，其中出于各种原因，设计要求自初始化而不是要求调用显式的模块初始化函数。

销毁互斥锁

互斥锁可以在解锁后立即销毁。但是，由于尝试销毁已锁定的互斥锁，或者另一个线程正尝试锁定的互斥锁，或者另一个线程正在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用的互斥锁，会导致未定义行为，因此必须注意确保没有其他线程可能引用该互斥锁。

健壮互斥锁

要求实现为进程共享属性设置为 `PTHREAD_PROCESS_SHARED` 的互斥锁提供健壮互斥锁。当进程共享属性设置为 `PTHREAD_PROCESS_PRIVATE` 时，允许但不要求实现提供健壮互斥锁。

另请参见

- `pthread_cond_clockwait()`
- `pthread_cond_timedwait()`
- `pthread_cond_wait()`

- `pthread_mutex_lock()`
- `pthread_mutex_unlock()`
- `pthread_once()`
- 基础定义, `<pthread.h>`

更改历史

源自 POSIX 线程扩展 (1003.1c-1995)。

信息性文本结束。

1.152. pthread_mutex_getprioceiling, pthread_mutex_setprioceiling

概要 (SYNOPSIS)

```
[RPP|TPP]
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict
                                  int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mute
                                  int prioceiling, int *restrict
```

描述 (DESCRIPTION)

`pthread_mutex_getprioceiling()` 函数应返回互斥锁的当前优先级上限。

`pthread_mutex_setprioceiling()` 函数应尝试锁定互斥锁，如同调用 `pthread_mutex_lock()` 一样，但锁定互斥锁的过程无需遵循优先级保护协议。在获取互斥锁后，它应更改互斥锁的优先级上限，然后如同调用 `pthread_mutex_unlock()` 一样释放互斥锁。当更改成功时，优先级上限的先前值应在 `old_ceiling` 中返回。

如果 `pthread_mutex_setprioceiling()` 函数失败，互斥锁优先级上限不应被更改。

返回值 (RETURN VALUE)

如果成功，`pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 函数应返回零；否则，应返回错误号以指示错误。

错误 (ERRORS)

这些函数应在以下情况失败：

- **[EINVAL]**

`mutex` 的协议属性是 PTHREAD_PRIO_NONE。

- **[EPERM]**

实现需要适当的权限来执行操作，而调用者没有适当的权限。

`pthread_mutex_setprioceiling()` 函数应在以下情况失败：

- **[EAGAIN]**

由于已超过 `mutex` 的最大递归锁定次数，无法获取互斥锁。

- **[EAGAIN]**

互斥锁是健壮互斥锁，且可用于拥有健壮互斥锁的系统资源将被超出。

- **[EDEADLK]**

互斥锁类型是 PTHREAD_MUTEX_ERRORCHECK，且当前线程已经拥有该互斥锁。

- **[EINVAL]**

互斥锁创建时协议属性值为 PTHREAD_PRIO_PROTECT，且调用线程的优先级高于互斥锁的当前优先级上限，并且实现在锁定互斥锁的过程中遵循优先级保护协议。

- **[ENOTRECOVERABLE]**

互斥锁是健壮互斥锁，且受互斥锁保护的状态不可恢复。

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，且包含先前拥有线程的进程在持有互斥锁时终止。

互斥锁应由调用线程获取，并由新所有者负责使状态一致（参见 `pthread_mutex_lock()`）。

`pthread_mutex_setprioceiling()` 函数可能在以下情况失败：

- **[EDEADLK]**

检测到死锁条件。

- **[EINVAL]**

由 `prioceiling` 请求的优先级超出范围。

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，且先前拥有线程在持有互斥锁时终止。互斥锁应由调用线程获取，并由新所有者负责使状态一致（参见 `pthread_mutex_lock()`）。

这些函数不应返回 [EINTR] 错误代码。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

无。

基本原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

另请参见 (SEE ALSO)

- `pthread_mutex_clocklock()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- XBD `<pthread.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 5

为与 POSIX 线程扩展对齐而包含。
标记为实时线程特性组的一部分。

Issue 6

`pthread_mutex_getprioceiling()` 和
`pthread_mutex_setprioceiling()` 函数被标记为线程和线程优先级保护选项的一部分。

删除了 [ENOSYS] 错误条件。

为与 IEEE Std 1003.1d-1999 对齐，将 `pthread_mutex_timedlock()` 函数添加到另请参见部分。

为与 ISO/IEC 9899:1999 标准对齐，向 `pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 原型添加了 **restrict** 关键字。

Issue 7

应用了 SD5-XSH-ERN-39。

应用了 Austin Group Interpretation 1003.1-2001 #052，添加 [EDEADLK] 作为"可能失败"错误。

应用了 SD5-XSH-ERN-158，更新错误部分以包含当 `mutex` 的协议属性为 PTHREAD_PRIO_NONE 时的"应失败"错误情况。

`pthread_mutex_getprioceiling()` 和

`pthread_mutex_setprioceiling()` 函数从线程选项移至需要支持健壮互斥锁优先级保护选项或非健壮互斥锁优先级保护选项。

更新描述和错误部分以正确处理所有各种互斥锁类型。

Issue 8

应用了 Austin Group Defect 354，为超出可用于拥有健壮互斥锁的系统资源添加 [EAGAIN] 错误。

1.153. `pthread_mutex_destroy`, `pthread_mutex_init`

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_destroy(pthread_mutex_t *mutex);
int pthread_mutex_init(pthread_mutex_t *restrict mutex,
                      const pthread_mutexattr_t *restrict attr)
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

DESCRIPTION

`pthread_mutex_destroy()` 函数应当销毁由 `mutex` 引用的互斥锁对象；该互斥锁对象实际上变为未初始化状态。实现可以使 `pthread_mutex_destroy()` 将由 `mutex` 引用的对象设置为无效值。

已销毁的互斥锁对象可以使用 `pthread_mutex_init()` 重新初始化；在销毁后其他方式引用该对象的结果是未定义的。

销毁一个已初始化且未锁定的互斥锁是安全的。尝试销毁一个已锁定的互斥锁，或另一个线程正在尝试锁定的互斥锁，或正在被另一个线程在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用的互斥锁，会导致未定义行为。

`pthread_mutex_init()` 函数应当使用由 `attr` 指定的属性初始化由 `mutex` 引用的互斥锁。如果 `attr` 为 NULL，则使用默认的互斥锁属性；其效果应与传递默认互斥锁属性对象的地址相同。成功初始化后，互斥锁的状态变为已初始化且未锁定。

进一步要求请参见 2.9.9 同步对象副本和替代映射。

尝试初始化一个已初始化的互斥锁会导致未定义行为。

在默认互斥锁属性合适的情况下，可以使用宏 PTHREAD_MUTEX_INITIALIZER 来初始化互斥锁。其效果应等同于通过调用 `pthread_mutex_init()` 并将参数 `attr` 指定为 NULL 进行动态初始化，但不执行错误检查。

如果传递给 `pthread_mutex_destroy()` 的 `mutex` 参数值不引用已初始化的互斥锁，则行为未定义。

如果传递给 `pthread_mutex_init()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，则行为未定义。

RETURN VALUE

如果成功，`pthread_mutex_destroy()` 和 `pthread_mutex_init()` 函数应返回零；否则，应返回错误码以指示错误。

ERRORS

`pthread_mutex_init()` 函数在以下情况下应失败：

- **[EAGAIN]**: 系统缺乏必要的资源（内存除外）来初始化另一个互斥锁。
- **[ENOMEM]**: 内存不足，无法初始化互斥锁。
- **[EPERM]**: 调用者没有执行该操作的权限。

`pthread_mutex_init()` 函数在以下情况下可能失败：

- **[EINVAL]**: 由 `attr` 引用的属性对象设置了健壮互斥锁属性，但未设置进程共享属性。

这些函数不应返回 [EINTR] 错误码。

以下各节为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

如果实现检测到传递给 `pthread_mutex_destroy()` 的 `mutex` 参数值不引用已初始化的互斥锁，建议函数失败并报告 [EINVAL] 错误。

如果实现检测到传递给 `pthread_mutex_destroy()` 或 `pthread_mutex_init()` 的 `mutex` 参数值引用已锁定的互斥锁或被另一个线程引用（例如，在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用）的互斥锁，或者检测到传递给 `pthread_mutex_init()` 的 `mutex` 参数值引用已初始化的互斥锁，建议函数失败并报告 [EBUSY] 错误。

如果实现检测到传递给 `pthread_mutex_init()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，建议函数失败并报告 [EINVAL] 错误。

可能的替代实现

本 POSIX.1-2024 卷支持互斥锁的几种替代实现。实现可以将锁直接存储在 `pthread_mutex_t` 类型的对象中。或者，实现可以将锁存储在堆中，并仅在互斥锁对象中存储指针、句柄或唯一 ID。任一实现都有其优势，或者在特定硬件配置下可能是必需的。为了能够编写对这种选择不变的便携代码，本 POSIX.1-2024 卷没有为此类型定义赋值或相等性，并使用术语“初始化”来加强（更具限制性的）概念，即锁实际上可能驻留在互斥锁对象本身中。

请注意，这排除了对互斥锁或条件变量类型的过度规范，并促使该类型的不透明性。

允许但不是要求 `pthread_mutex_destroy()` 在互斥锁中存储非法值。这可能有助于检测试图锁定（或以其他方式引用）已销毁互斥锁的错误程序。

支持错误检查与性能之间的权衡

许多可能发生的错误情况不需要实现检测，以便让实现根据其特定应用程序和执行环境的需要，在性能与错误检查程度之间进行权衡。作为一般规则，由系统引起的条件（如内存不足）要求检测，但由错误编码的应用程序引起的条件（如未能提供足够的同步以防止互斥锁在使用时被删除）被指定为导致未定义行为。

因此，可以实现广泛的实现范围。例如，用于应用程序调试的实现可以实现所有错误检查，但在嵌入式计算机上在非常严格的性能约束下运行单个、经证明正确的应用程序的实现可能实现最少的检查。实现甚至可能以两个版本提供，类似于编译器提供的选项：一个全检查但较慢的版本；和一个有限检查但更快的版本。禁止这种可选性对用户来说是一种不友好的做法。

通过仔细将"未定义行为"的使用限制为错误（错误编码的）应用程序可能做的事情，并通过定义资源不可用错误是强制性的，本 POSIX.1-2024 卷确保完全符合规范的应用程序在全部实现范围内是可移植的，同时不强制所有实现为检查正确程序永远不会做的众多事情而添加开销。当行为未定义时，对于确实检测到该条件的实现，没有指定要返回的错误码。这是因为未定义行为意味着任何事情都可能发生，包括返回任何值（这可能恰好是有效但不同的错误码）。但是，由于错误码可能对应用程序开发人员在应用程序开发期间诊断问题时有帮助，在基本原理中提出了一个建议：如果他们的实现确实检测到该条件，实现者应返回特定的错误码。

为什么没有定义限制

曾考虑为互斥锁和条件变量的最大数量定义符号，但被拒绝，因为这些对象的数量可能动态变化。此外，许多实现将这些对象放在应用程序内存中；因此，没有明确的限制。

互斥锁和条件变量的静态初始化程序

为静态分配的同步对象提供静态初始化允许具有私有静态同步变量的模块避免运行时初始化测试和开销。此外，它简化了自初始化模块的编码。此类模块在 C 库中很常见，其中出于各种原因，设计要求自初始化，而不是要求调用显式的模块初始化函数。静态初始化的示例如下。

没有静态初始化，自初始化例程 `foo()` 可能如下所示：

```
static pthread_once_t foo_once = PTHREAD_ONCE_INIT;
static pthread_mutex_t foo_mutex;

void foo_init()
{
    pthread_mutex_init(&foo_mutex, NULL);
}

void foo()
{
    pthread_once(&foo_once, foo_init);
    pthread_mutex_lock(&foo_mutex);
    /* 执行工作。 */
    pthread_mutex_unlock(&foo_mutex);
}
```

使用静态初始化，相同的例程可以编码如下：

```
static pthread_mutex_t foo_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
void foo()
{
    pthread_mutex_lock(&foo_mutex);
    /* 执行工作。 */
    pthread_mutex_unlock(&foo_mutex);
}
```

请注意，静态初始化既消除了在 `pthread_once()` 内部进行初始化测试的需要，也消除了获取 `foo_mutex` 以了解要传递给 `pthread_mutex_lock()` 或 `pthread_mutex_unlock()` 的地址的需要。

因此，在所有系统上为初始化静态对象而编写的 C 代码更简单，并且在大量系统上也更快；那些（整个）同步对象可以存储在应用程序内存中的系统。

然而，对于需要将互斥锁分配出特殊内存的机器，可能会提出锁定性能问题。此类机器实际上必须让互斥锁，可能还有条件变量包含指向实际硬件锁的指针。为了使静态初始化在此类机器上工作，`pthread_mutex_lock()` 还必须测试实际锁的指针是否已分配。如果没有，`pthread_mutex_lock()` 在使用前必须初始化它。此类资源的保留可以在程序加载时进行，因此没有向互斥锁锁定和条件变量等待添加返回码以指示未能完成初始化。

`pthread_mutex_lock()` 中的此运行时测试起初似乎是额外的工作；需要额外的测试来查看指针是否已初始化。在大多数机器上，这实际上将实现为获取指针、将指针与零进行测试，然后在指针已初始化时使用指针。虽然测试似乎会增加额外的工作，但测试寄存器的额外努力通常可以忽略不计，因为实际上没有进行额外的内存引用。随着越来越多的机器提供缓存，真正的开销是内存引用，而不是执行的指令。

或者，根据机器架构，通常有方法可以在最重要的情况下消除所有开销：在已初始化锁之后发生的锁操作上。这可以通过将更多开销转移到较不频繁的操作来实现：初始化。由于互斥锁的线外分配也意味着必须取消引用地址以找到实际的锁，一种广泛适用的技术是让静态初始化为该地址存储一个伪值；特别是，会导致机器故障发生的地址。当在首次尝试锁定此类互斥锁时发生此类故障时，可以进行有效性检查，然后可以填入实际锁的正确地址。后续的锁操作不会产生额外开销，因为它们不会“故障”。这仅仅是可用于支持静态初始化而不对锁获取性能产生不利影响的技术之一。无疑还有其他高度依赖于机器的技术。

因此，对于进行线外互斥锁分配的机器，锁定开销对于隐式初始化的模块是相似的，而对于完全内联进行互斥锁分配的模块则有所改善。内联情况因此变得更快，而线外情况不会明显更差。

除了此类机器的锁定性能问题外，还有一个担忧是线程可能在尝试完成静态分配的互斥锁初始化时为初始化锁序列化竞争。（此类完成通常涉及获取内部锁、分配结构、将指向结构的指针存储在互斥锁中，并释放内部锁。）首先，许多实

现会通过对互斥锁地址进行哈希来减少此类序列化。其次，此类序列化只能发生有限的次数。特别是，它最多可能发生与静态分配同步对象数量相同的次数。动态分配的对象仍将通过 `pthread_mutex_init()` 或 `pthread_cond_init()` 初始化。

最后，如果以上线外分配的优化技术都没有为某个实现上的应用程序提供足够的性能，应用程序可以通过使用相应的 `pthread_*_init()` 函数显式初始化所有同步对象来完全避免静态初始化，这些函数被所有实现支持。实现还可以记录权衡并建议哪种初始化技术对于该特定实现更有效。

销毁互斥锁

互斥锁可以在解锁后立即销毁。但是，由于尝试销毁已锁定的互斥锁，或另一个线程正在尝试锁定的互斥锁，或正在被另一个线程在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中使用的互斥锁，会导致未定义行为，必须注意确保没有其他线程可能在引用该互斥锁。

健壮互斥锁

对于进程共享属性设置为 `PTHREAD_PROCESS_SHARED` 的互斥锁，要求实现提供健壮互斥锁。当进程共享属性设置为 `PTHREAD_PROCESS_PRIVATE` 时，允许但不是要求实现提供健壮互斥锁。

FUTURE DIRECTIONS

无。

SEE ALSO

`pthread_mutex_getprioceiling()` , `pthread_mutexattr_getrobust()` ,
`pthread_mutex_clockclock()` , `pthread_mutex_lock()` ,
`pthread_mutexattr_getpshared()`

XBD `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含在内。

Issue 6

`pthread_mutex_destroy()` 和 `pthread_mutex_init()` 函数被标记为线程选项的一部分。

为了与 IEEE Std 1003.1-2001 对齐，在 SEE ALSO 部分添加了 `pthread_mutex_timedlock()` 函数。

1.154. pthread_mutex_lock

概要

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

描述

通过调用 `pthread_mutex_lock()` 来锁定 `mutex` 引用的互斥锁对象，该调用返回零或 [EOWNERDEAD]。如果互斥锁已被其他线程锁定，调用线程将被阻塞，直到互斥锁变为可用。此操作返回时，`mutex` 引用的互斥锁对象将处于锁定状态，调用线程为其所有者。如果线程尝试重新锁定它已经锁定的互斥锁，`pthread_mutex_lock()` 的行为将如下表 **重新锁定** 列所述。如果线程尝试解锁它未锁定的互斥锁或已解锁的互斥锁，`pthread_mutex_unlock()` 的行为将如下表 **非所有者解锁** 列所述。

互斥锁类型	健壮性	重新锁定	非所有者解锁
NORMAL	非健壮	死锁	未定义行为
NORMAL	健壮	死锁	返回错误
ERRORCHECK	任意	返回错误	返回错误
RECURSIVE	任意	递归（见下文）	返回错误
DEFAULT	非健壮	未定义行为 +	未定义行为 +
DEFAULT	健壮	未定义行为 +	返回错误

+ 如果互斥锁类型为 PTHREAD_MUTEX_DEFAULT，`pthread_mutex_lock()` 的行为可能对应上表中其他三种标准互斥锁类型之一。如果不对应这三种类型中的任何一种，则标记为 + 的情况下的行为是未定义的。

当表中表示递归行为时，互斥锁应维护锁定计数概念。当线程首次成功获取互斥锁时，锁定计数应设置为一。每次线程重新锁定此互斥锁时，锁定计数应加一。每次线程解锁互斥锁时，锁定计数应减一。当锁定计数达到零时，互斥锁应变为可供其他线程获取的状态。

`pthread_mutex_trylock()` 函数应等效于 `pthread_mutex_lock()`，但如果 *mutex* 引用的互斥锁对象当前被锁定（由任何线程，包括当前线程锁定），调用应立即返回。如果互斥锁类型为 PTHREAD_MUTEX_RECURSIVE 且互斥锁当前由调用线程拥有，互斥锁锁定计数应加一，`pthread_mutex_trylock()` 函数应立即返回成功。

`pthread_mutex_unlock()` 函数应释放 *mutex* 引用的互斥锁对象。互斥锁的释放方式取决于互斥锁的类型属性。如果在调用 `pthread_mutex_unlock()` 时有线程阻塞在 *mutex* 引用的互斥锁对象上，导致互斥锁变为可用，调度策略应决定哪个线程应获取该互斥锁。

（对于 PTHREAD_MUTEX_RECURSIVE 互斥锁，当计数达到零且调用线程不再拥有此互斥锁的任何锁时，互斥锁应变为可用。）

如果信号被传递给等待互斥锁的线程，从信号处理程序返回后，线程应恢复等待互斥锁，就像未被中断一样。

如果 *mutex* 是健壮互斥锁且包含拥有线程的进程在持有互斥锁时终止，调用 `pthread_mutex_lock()` 应返回错误值 [EOWNERDEAD]。如果 *mutex* 是健壮互斥锁且拥有线程在持有互斥锁时终止，即使拥有线程所在的进程尚未终止，调用 `pthread_mutex_lock()` 也可能返回错误值 [EOWNERDEAD]。在这些情况下，互斥锁应被调用线程锁定，但其保护的状态被标记为不一致。应用程序应确保状态变得一致以供重用，完成后调用 `pthread_mutex_consistent()`。如果应用程序无法恢复状态，它应在未先调用 `pthread_mutex_consistent()` 的情况下解锁互斥锁，之后该互斥锁被标记为永久不可用。

如果 *mutex* 不引用已初始化的互斥锁对象，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 的行为是未定义的。

返回值

如果成功，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数在以下情况下应失败：

- **[EAGAIN]**

- 无法获取互斥锁，因为 *mutex* 的递归锁最大数量已被超过。
- 互斥锁是健壮互斥锁，且可用的健壮互斥锁拥有的系统资源将被超过。

- **[EINVAL] [RPP|TPP]**

- *mutex* 是使用协议属性值 PTHREAD_PRIO_PROTECT 创建的，且调用线程的优先级高于互斥锁的当前优先级上限。

- **[ENOTRECOVERABLE]**

- 互斥锁保护的状态不可恢复。

- **[EOWNERDEAD]**

- 互斥锁是健壮互斥锁，且包含先前拥有线程的进程在持有互斥锁时终止。
互斥锁应由调用线程获取，并由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数在以下情况下应失败：

- **[EDEADLK]**

- 互斥锁类型为 PTHREAD_MUTEX_ERRORCHECK，且当前线程已拥有该互斥锁。

`pthread_mutex_trylock()` 函数在以下情况下应失败：

- **[EBUSY]**

- 无法获取 *mutex*，因为它已被锁定。

`pthread_mutex_unlock()` 函数在以下情况下应失败：

- **[EPERM]**

- 互斥锁类型为 PTHREAD_MUTEX_ERRORCHECK 或 PTHREAD_MUTEX_RECURSIVE，或者互斥锁是健壮互斥锁，且当前线程不拥有该互斥锁。

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数在以下情况下可能失败：

- **[EOWNERDEAD]**

- 互斥锁是健壮互斥锁，且先前拥有线程在持有互斥锁时终止。互斥锁应由调用线程获取，并由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数在以下情况下可能失败：

- **[EDEADLK]**
- 检测到死锁条件。

这些函数不应返回错误代码 [EINTR]。

以下章节为信息性内容。

示例

无。

应用程序用法

假设非零返回值为错误的应用程序需要更新才能与健壮互斥锁一起使用，因为线程获取保护当前不一致状态的互斥锁时的有效返回值是 [EOWNERDEAD]。由于排除了出现此类错误的可能性而不检查错误返回值的应用程序不应使用健壮互斥锁。如果应用程序应与普通和健壮互斥锁一起工作，它应检查所有返回值的错误条件，并在必要时采取适当操作。

原理

互斥锁对象旨在作为底层原语，可基于此构建其他线程同步函数。因此，互斥锁的实现应尽可能高效，这对接口上可用的功能有影响。

互斥锁函数和互斥锁属性的特定默认设置是出于不排除互斥锁锁定和解锁的快速内联实现的愿望而驱动的。

由于大多数属性仅在线程即将被阻塞时才需要检查，因此使用属性不会减慢（常见的）互斥锁锁定情况。

同样，虽然能够提取互斥锁所有者的线程 ID 可能是可取的，但这需要在锁定每个互斥锁时存储当前线程 ID，这可能导致不可接受的性能开销。类似参数适用于 `mutex_tryunlock` 操作。

有关扩展互斥锁类型的进一步原理，请参见 XRAT 线程扩展。

如果实现检测到 `mutex` 参数指定的值不引用已初始化的互斥锁对象，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

另请参阅

`pthread_mutex_clocklock()` , `pthread_mutex_consistent()` ,
`pthread_mutex_destroy()` , `pthread_mutexattr_getrobust()`

XBD 4.15.2 内存同步, `<pthread.h>`

变更历史

首次在 Issue 5 中发布。为与 POSIX 线程扩展对齐而包含。

Issue 6

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和
`pthread_mutex_unlock()` 函数被标记为线程选项的一部分。

以下对 POSIX 实现的新要求源自与单一 UNIX 规范的对齐：

- 定义了尝试重新锁定互斥锁时的行为。

`pthread_mutex_timedlock()` 函数被添加到另请参阅部分，以便与 IEEE Std 1003.1d-1999 对齐。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/98，更新错误部分，使 [EDEADLK] 错误包括死锁条件检测。原理部分也被重新措辞，以考虑非 XSI 一致系统。

Issue 7

应用 SD5-XSH-ERN-43，将 [EINVAL] 错误的“应失败”情况标记为依赖于线程优先级保护选项。

根据 The Open Group Technical Standard, 2006, Extended API Set Part 3 进行更改。

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和
`pthread_mutex_unlock()` 函数从线程选项移至基础。

以下扩展互斥锁类型从 XSI 选项移至基础：

PTHREAD_MUTEX_NORMAL
PTHREAD_MUTEX_ERRORCHECK
PTHREAD_MUTEX_RECURSIVE
PTHREAD_MUTEX_DEFAULT

更新描述以阐明 *mutex* 不引用已初始化互斥锁时的行为。

更新错误部分以正确处理所有各种互斥锁类型。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0461 [121],
XSH/TC1-2008/0462 [92,428], 和 XSH/TC1-2008/0463 [121]。

Issue 8

应用 Austin Group Defect 354，为超过可用的健壮互斥锁拥有的系统资源添加 [EAGAIN] 错误。

应用 Austin Group Defect 1115，将"the thread"更改为"the calling thread"。

1.155. `pthread_mutex_getprioceiling`, `pthread_mutex_setprioceiling`

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_getprioceiling(const pthread_mutex_t *restrict
                                  int *restrict prioceiling);

int pthread_mutex_setprioceiling(pthread_mutex_t *restrict mute
                                  int prioceiling, int *restrict
```

DESCRIPTION

`pthread_mutex_getprioceiling()` 函数应返回互斥锁的当前优先级上限。

`pthread_mutex_setprioceiling()` 函数应尝试锁定互斥锁，如同调用 `pthread_mutex_lock()` 一样，但锁定互斥锁的过程不需要遵循优先级保护协议。在获取互斥锁后，它应更改互斥锁的优先级上限，然后如同调用 `pthread_mutex_unlock()` 一样释放互斥锁。当更改成功时，优先级上限的前一个值应返回到 `old_ceiling` 中。

如果 `pthread_mutex_setprioceiling()` 函数失败，互斥锁优先级上限不应被更改。

RETURN VALUE

如果成功，`pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS

这些函数在以下情况下应失败：

- **[EINVAL]**

`mutex` 的协议属性是 PTHREAD_PRIO_NONE。

- **[EPERM]**

实现需要适当的权限来执行操作，但调用者没有适当的权限。

`pthread_mutex_setprioceiling()` 函数在以下情况下应失败：

- **[EAGAIN]**

由于 `mutex` 的递归锁最大数量已被超过，无法获取互斥锁。

- **[EAGAIN]**

互斥锁是健壮互斥锁，并且可用的健壮互斥锁拥有的系统资源将被超出。

- **[EDEADLK]**

互斥锁类型是 PTHREAD_MUTEX_ERRORCHECK，并且当前线程已经拥有该互斥锁。

- **[EINVAL]**

互斥锁创建时协议属性值为 PTHREAD_PRIO_PROTECT，并且调用线程的优先级高于互斥锁的当前优先级上限，且实现在锁定互斥锁的过程中遵循优先级保护协议。

- **[ENOTRECOVERABLE]**

互斥锁是健壮互斥锁，并且互斥锁保护的状态不可恢复。

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，包含前一个拥有线程的进程在持有互斥锁时终止。

调用线程应获取互斥锁，由新所有者负责使状态一致（参见 `pthread_mutex_lock()`）。

`pthread_mutex_setprioceiling()` 函数在以下情况下可能失败：

- **[EDEADLK]**

检测到死锁条件。

- **[EINVAL]**

`prioceiling` 请求的优先级超出范围。

- **[EOWNERDEAD]**

互斥锁是健壮互斥锁，前一个拥有线程在持有互斥锁时终止。调用线程应获取互斥锁，由新所有者负责使状态一致（参见 `pthread_mutex_lock()`）。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_mutex_clocklock()`
- `pthread_mutex_destroy()`
- `pthread_mutex_lock()`
- `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。标记为实时线程特性组的一部分。

Issue 6

- `pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 函数标记为线程和线程优先级保护选项的一部分。
- 移除了 [ENOSYS] 错误条件。

- 为与 IEEE Std 1003.1d-1999 对齐，在 SEE ALSO 部分添加了 `pthread_mutex_timedlock()` 函数。
- 为与 ISO/IEC 9899:1999 标准对齐，在 `pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 原型中添加了 `restrict` 关键字。

Issue 7

- 应用了 SD5-XSH-ERN-39。
- 应用了 Austin Group Interpretation 1003.1-2001 #052，添加 [EDEADLK] 作为"可能失败"错误。
- 应用了 SD5-XSH-ERN-158，更新 ERRORS 部分以包含当 `mutex` 的协议属性为 PTHREAD_PRIO_NONE 时的"应失败"错误情况。
- `pthread_mutex_getprioceiling()` 和 `pthread_mutex_setprioceiling()` 函数从线程选项移至需要支持健壮互斥锁优先级保护选项或非健壮互斥锁优先级保护选项。
- 更新了 DESCRIPTION 和 ERRORS 部分，以正确考虑所有各种互斥锁类型。

Issue 8

- 应用了 Austin Group Defect 354，添加了超出可用的健壮互斥锁拥有的系统资源时的 [EAGAIN] 错误。
-

1.156. pthread_mutex_trylock - 尝试锁定互斥锁

概要

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

描述

由 `mutex` 引用的互斥锁对象应通过调用 `pthread_mutex_lock()` 来锁定，该调用返回零或 `[EOWNERDEAD]`。如果互斥锁已被另一个线程锁定，调用线程将阻塞直到互斥锁变得可用。此操作返回时，`mutex` 引用的互斥锁对象处于锁定状态，调用线程为其所有者。如果线程尝试重新锁定它已经锁定的互斥锁，`pthread_mutex_lock()` 的行为应如下表 **重新锁定** 列所述。如果线程尝试解锁它没有锁定的互斥锁或未锁定的互斥锁，`pthread_mutex_unlock()` 的行为应如下表 **非所有者解锁** 列所述。

互斥锁类型	健壮性	重新锁定	非所有者解锁
NORMAL	非健壮	死锁	未定义行为
NORMAL	健壮	死锁	返回错误
ERRORCHECK	任意	返回错误	返回错误
RECURSIVE	任意	递归（见下文）	返回错误
DEFAULT	非健壮	未定义行为 †	未定义行为 †
DEFAULT	健壮	未定义行为 †	返回错误

† 如果互斥锁类型是 `PTHREAD_MUTEX_DEFAULT`，`pthread_mutex_lock()` 的行为可能对应于上表中所述的三种其他标准互斥锁类型之一。如果它不对应于这三种类型之一，则标记为 † 的情况下的行为是未定义的。

在表指示递归行为的情况下，互斥锁应保持锁计数概念。当线程第一次成功获取互斥锁时，锁计数应设置为一。每次线程重新锁定此互斥锁时，锁计数应加一。每次线程解锁互斥锁时，锁计数应减一。当锁计数达到零时，互斥锁应对其他线程可用以获取。

`pthread_mutex_trylock()` 函数应等价于 `pthread_mutex_lock()`，但如果 `mutex` 引用的互斥锁对象当前被锁定（由任何线程，包括当前线程），调用应立即返回。如果互斥锁类型是 `PTHREAD_MUTEX_RECURSIVE` 且互斥锁当前由调用线程拥有，互斥锁锁计数应加一，`pthread_mutex_trylock()` 函数应立即返回成功。

`pthread_mutex_unlock()` 函数应释放 `mutex` 引用的互斥锁对象。释放互斥锁的方式取决于互斥锁的类型属性。如果在调用 `pthread_mutex_unlock()` 时有线程在 `mutex` 引用的互斥锁对象上阻塞，导致互斥锁变得可用，调度策略应确定哪个线程应获取互斥锁。

（在 `PTHREAD_MUTEX_RECURSIVE` 互斥锁的情况下，当计数达到零且调用线程不再拥有此互斥锁的任何锁时，互斥锁应变为可用。）

如果信号传递给等待互斥锁的线程，从信号处理程序返回后，线程应恢复等待互斥锁，就像它未被中断一样。

如果 `mutex` 是健壮互斥锁且拥有互斥锁的进程在保持互斥锁时终止，调用 `pthread_mutex_lock()` 应返回错误值 `[EOWNERDEAD]`。如果 `mutex` 是健壮互斥锁且拥有线程在保持互斥锁时终止，即使拥有线程所在的进程尚未终止，调用 `pthread_mutex_lock()` 也可能返回错误值 `[EOWNERDEAD]`。在这些情况下，互斥锁应由调用线程锁定，但它保护的状态被标记为不一致。应用程序应确保状态变为一致以供重用，完成后调用 `pthread_mutex_consistent()`。如果应用程序无法恢复状态，应在未调用 `pthread_mutex_consistent()` 的情况下解锁互斥锁，此后互斥锁被标记为永久不可用。

如果 `mutex` 不引用已初始化的互斥锁对象，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 的行为是未定义的。

返回值

如果成功，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数应在以下情况下失败：

- `[EAGAIN]` — 互斥锁无法获取，因为超过了 `mutex` 的递归锁最大次数。
- `[EAGAIN]` — 互斥锁是健壮互斥锁，可用的健壮互斥锁系统资源将被超出。
- `[EINVAL]` — `[RPP|TPP]` `mutex` 是用协议属性值为 `PTHREAD_PRIO_PROTECT` 创建的，调用线程的优先级高于互斥锁的当前优先级上限。
- `[ENOTRECOVERABLE]` — 互斥锁保护的状态不可恢复。
- `[EOWNERDEAD]` — 互斥锁是健壮互斥锁，包含先前拥有线程的进程在保持互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数应在以下情况下失败：

- `[EDEADLK]` — 互斥锁类型是 `PTHREAD_MUTEX_ERRORCHECK` 且当前线程已经拥有互斥锁。

`pthread_mutex_trylock()` 函数应在以下情况下失败：

- `[EBUSY]` — `mutex` 无法获取，因为它已被锁定。

`pthread_mutex_unlock()` 函数应在以下情况下失败：

- `[EPERM]` — 互斥锁类型是 `PTHREAD_MUTEX_ERRORCHECK` 或 `PTHREAD_MUTEX_RECURSIVE`，或互斥锁是健壮互斥锁，且当前线程不拥有互斥锁。

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数可能在以下情况下失败：

- `[EOWNERDEAD]` — 互斥锁是健壮互斥锁，先前拥有线程在保持互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数可能在以下情况下失败：

- `[EDEADLK]` — 检测到死锁条件。

这些函数不应返回 `[EINTR]` 错误代码。

应用程序用法

假设非零返回值是错误的应用程序将需要更新以与健壮互斥锁一起使用，因为线程获取保护当前不一致状态的互斥锁的有效返回是 `[EOWNERDEAD]`。由于排除了此类错误出现的可能性而不检查错误返回的应用程序不应使用健壮互斥锁。如果应用程序应该与普通和健壮互斥锁一起工作，它应检查所有错误条件的返回值，并在必要时采取适当行动。

基本原理

互斥锁对象旨在作为可以构建其他线程同步函数的低级原语。因此，互斥锁的实现应尽可能高效，这对接口上可用功能有影响。

互斥锁函数和互斥锁属性的特定默认设置受到不排除快速、内联互斥锁锁定和解锁实现的愿望的激励。

由于大多数属性只需要在线程将被阻塞时检查，属性的使用不会减慢（常见）互斥锁锁定情况。

同样，虽然能够提取互斥锁拥有者的线程 ID 可能是可取的，但这需要在锁定每个互斥锁时存储当前线程 ID，这可能导致不可接受的级别开销。类似的论点适用于 `mutex_tryunlock` 操作。

有关扩展互斥锁类型的进一步基本原理，请参见 X RAT 线程扩展。

如果实现检测到 `mutex` 参数指定的值不引用已初始化的互斥锁对象，建议函数应失败并报告 `[EINVAL]` 错误。

另见

- `pthread_mutex_clocklock()`
- `pthread_mutex_consistent()`
- `pthread_mutex_destroy()`
- `pthread_mutexattr_getrobust()`
- 4.15.2 内存同步
- `<pthread.h>`

变更历史

首次在 Issue 5 中发布。包含用于与 POSIX 线程扩展对齐。

Issue 6

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数被标记为线程选项的一部分。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 重新锁定互斥锁时的行为被定义。

`pthread_mutex_timedlock()` 函数被添加到另见部分，以便与 IEEE Std 1003.1d-1999 对齐。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/98，更新错误部分，使 `[EDEADLK]` 错误包含死锁条件的检测。基本原理部分也被重新措辞，以考虑非 XSI 符合系统。

Issue 7

应用 SD5-XSH-ERN-43，将 `[EINVAL]` 错误的"应失败"情况标记为依赖于线程优先级保护选项。

从 The Open Group 技术标准 2006 扩展 API 集第 3 部分进行更改。

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数从线程选项移动到基本。

以下扩展互斥锁类型从 XSI 选项移动到基本：

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

描述被更新以澄清当 `mutex` 不引用已初始化互斥锁时的行为。

错误部分被更新以正确处理所有各种互斥锁类型。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0461 [121]、XSH/TC1-2008/0462 [92,428] 和 XSH/TC1-2008/0463 [121]。

Issue 8

应用 Austin Group Defect 354，为超出可用的健壮互斥锁系统资源添加
[EAGAIN] 错误。

应用 Austin Group Defect 1115，将"该线程"更改为"调用线程"。

1.157. `pthread_mutex_lock`, `pthread_mutex_trylock`, `pthread_mutex_unlock`

SYNOPSIS

```
#include <pthread.h>

int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

DESCRIPTION

通过调用 `pthread_mutex_lock()` 并返回零或 `[EOWNERDEAD]`，`mutex` 引用的互斥锁对象应当被锁定。如果互斥锁已被另一个线程锁定，调用线程应当阻塞，直到互斥锁变为可用。此操作应当在 `mutex` 引用的互斥锁对象处于锁定状态且调用线程为其所有者时返回。如果一个线程尝试重新锁定一个已经锁定的互斥锁，`pthread_mutex_lock()` 的行为应如下表 **重新锁定** 列所述。如果一个线程尝试解锁一个未锁定的或未由其锁定的互斥锁，`pthread_mutex_unlock()` 的行为应如下表 **非所有者解锁** 列所述。

互斥锁类型	健壮性	重新锁定	非所有者解锁
NORMAL	非健壮	死锁	未定义行为
NORMAL	健壮	死锁	返回错误
ERRORCHECK	任意	返回错误	返回错误
RECURSIVE	任意	递归（见下文）	返回错误
DEFAULT	非健壮	未定义行为 †	未定义行为 †
DEFAULT	健壮	未定义行为 †	返回错误

† 如果互斥锁类型为 `PTHREAD_MUTEX_DEFAULT`，`pthread_mutex_lock()` 的行为可能对应上表中所述的三种其他标准互斥锁类型之一。如果它不对应这三种类型之一，则标记为 † 的行为是未定义的。

当表指示递归行为时，互斥锁应维护锁计数的概念。当线程首次成功获取互斥锁时，锁计数应设置为一。每次线程重新锁定此互斥锁时，锁计数应增加一。每次线程解锁互斥锁时，锁计数应减少一。当锁计数达到零时，互斥锁应变为可供其他线程获取。

`pthread_mutex_trylock()` 函数应等同于 `pthread_mutex_lock()`，但如果 `mutex` 引用的互斥锁对象当前被锁定（由任何线程，包括当前线程），调用应立即返回。如果互斥锁类型为 `PTHREAD_MUTEX_RECURSIVE` 且互斥锁当前由调用线程拥有，互斥锁锁计数应增加一，且 `pthread_mutex_trylock()` 函数应立即返回成功。

`pthread_mutex_unlock()` 函数应释放 `mutex` 引用的互斥锁对象。释放互斥锁的方式取决于互斥锁的类型属性。当调用 `pthread_mutex_unlock()` 时，如果有线程阻塞在 `mutex` 引用的互斥锁对象上，导致互斥锁变为可用，调度策略应决定哪个线程应获取互斥锁。

（在 `PTHREAD_MUTEX_RECURSIVE` 互斥锁的情况下，当计数达到零且调用线程不再对此互斥锁有任何锁定时，互斥锁应变为可用。）

如果信号被传递给等待互斥锁的线程，从信号处理程序返回后，线程应恢复等待互斥锁，就像未被中断一样。

如果 `mutex` 是健壮互斥锁且拥有互斥锁的线程的进程在持有互斥锁时终止，调用 `pthread_mutex_lock()` 应返回错误值 `[EOWNERDEAD]`。如果 `mutex` 是健壮互斥锁且拥有互斥锁的线程在持有互斥锁时终止，即使拥有线程所在的进程尚未终止，调用 `pthread_mutex_lock()` 也可能返回错误值 `[EOWNERDEAD]`。在这些情况下，互斥锁应由调用线程锁定，但它保护的状态被标记为不一致。应用程序应确保状态变得一致以便重用，并在完成时调用 `pthread_mutex_consistent()`。如果应用程序无法恢复状态，它应在未事先调用 `pthread_mutex_consistent()` 的情况下解锁互斥锁，此后互斥锁被标记为永久不可用。

如果 `mutex` 不引用已初始化的互斥锁对象，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 的行为是未定义的。

RETURN VALUE

如果成功，`pthread_mutex_lock()`、`pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数应返回零；否则，应返回错误号以指示错误。

ERRORS

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数应在以下情况下失败：

- **[EAGAIN]** 无法获取互斥锁，因为已超过 `mutex` 的最大递归锁定次数。
- **[EAGAIN]** 互斥锁是健壮互斥锁，且可用于拥有健壮互斥锁的系统资源将被超出。
- **[EINVAL]** `[RPP|TPP]` `mutex` 是用协议属性值为 `PTHREAD_PRIO_PROTECT` 创建的，且调用线程的优先级高于互斥锁的当前优先级上限。
- **[ENOTRECOVERABLE]** 互斥锁保护的状态不可恢复。
- **[EOWNERDEAD]** 互斥锁是健壮互斥锁，且包含先前拥有线程的进程在持有互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数应在以下情况下失败：

- **[EDEADLK]** 互斥锁类型为 `PTHREAD_MUTEX_ERRORCHECK` 且当前线程已拥有互斥锁。

`pthread_mutex_trylock()` 函数应在以下情况下失败：

- **[EBUSY]** 无法获取 `mutex`，因为它已被锁定。

`pthread_mutex_unlock()` 函数应在以下情况下失败：

- **[EPERM]** 互斥锁类型为 `PTHREAD_MUTEX_ERRORCHECK` 或 `PTHREAD_MUTEX_RECURSIVE`，或互斥锁是健壮互斥锁，且当前线程不拥有互斥锁。

`pthread_mutex_lock()` 和 `pthread_mutex_trylock()` 函数可能在以下情况下失败：

- **[EOWNERDEAD]** 互斥锁是健壮互斥锁，且先前的拥有线程在持有互斥锁时终止。互斥锁应由调用线程获取，由新所有者负责使状态一致。

`pthread_mutex_lock()` 函数可能在以下情况下失败：

- **[EDEADLK]** 检测到死锁条件。

这些函数不应返回 `[EINTR]` 错误代码。

EXAMPLES

无。

APPLICATION USAGE

已假设非零返回值是错误的应用程序将需要更新以与健壮互斥锁一起使用，因为线程获取保护当前不一致状态的互斥锁的有效返回是 `[EOWNERDEAD]`。由于排除了此类错误出现的可能性而不检查错误返回的应用程序不应使用健壮互斥锁。如果应用程序应该与普通和健壮互斥锁一起工作，它应检查所有返回值的错误条件，并在必要时采取适当行动。

RATIONALE

互斥锁对象旨在作为构建其他线程同步函数的低级原语。因此，互斥锁的实现应尽可能高效，这对接口可用的功能产生影响。

互斥锁函数和互斥锁属性的特定默认设置是出于不排除互斥锁锁定和解锁的快速内联实现的愿望。

由于大多数属性只需要在线程将被阻塞时检查，属性的使用不会减慢（常见的）互斥锁锁定情况。

同样，虽然能够提取互斥锁拥有者的线程 ID 可能是可取的，但这需要在每个互斥锁被锁定时存储当前线程 ID，这可能导致不可接受的开销级别。类似的论证适用于 `mutex_tryunlock` 操作。

有关扩展互斥锁类型的基本原理，请参阅 X RAT [线程扩展](#)。

如果实现检测到 `mutex` 参数指定的值不引用已初始化的互斥锁对象，建议函数应失败并报告 `[EINVAL]` 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_mutex_clocklock()`
- `pthread_mutex_consistent()`

- `pthread_mutex_destroy()`
- `pthread_mutexattr_getrobust()`
- XBD 4.15.2 内存同步
-

CHANGE HISTORY

Issue 6

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数被标记为线程选项的一部分。

POSIX 实现的以下新要求源于与单一 UNIX 规范的对齐：

- 定义了尝试重新锁定互斥锁时的行为。

为与 IEEE Std 1003.1d-1999 对齐，在 SEE ALSO 部分添加了 `pthread_mutex_timedlock()` 函数。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/98，更新 ERRORS 部分，使 `[EDEADLK]` 错误包括死锁条件的检测。RATIONALE 部分也被重新措辞，以考虑不符合 XSI 的系统。

Issue 7

应用 SD5-XSH-ERN-43，将 `[EINVAL]` 错误的“应失败”情况标记为依赖于线程优先级保护选项。

从 The Open Group Technical Standard, 2006, Extended API Set Part 3 进行了更改。

`pthread_mutex_lock()` 、 `pthread_mutex_trylock()` 和 `pthread_mutex_unlock()` 函数从线程选项移动到基础。

以下扩展互斥锁类型从 XSI 选项移动到基础：

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

更新 DESCRIPTION 以阐明当 `mutex` 不引用已初始化互斥锁时的行为。

更新 ERRORS 部分以正确考虑所有各种互斥锁类型。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0461 [121], XSH/TC1-2008/0462 [92,428], 和 XSH/TC1-2008/0463 [121]。

Issue 8

应用 Austin Group Defect 354, 添加超出可用于拥有的健壮互斥锁的系统资源的 **[EAGAIN]** 错误。

应用 Austin Group Defect 1115, 将"该线程"更改为"调用线程"。

1.158. pthread_mutexattr_destroy

概要

```
#include <pthread.h>

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
```

描述

`pthread_mutexattr_destroy()` 函数应销毁一个互斥锁属性对象；该对象实际上变为未初始化状态。实现可能导致 `pthread_mutexattr_destroy()` 将 `attr` 引用的对象设置为无效值。

已销毁的 `attr` 属性对象可以使用 `pthread_mutexattr_init()` 重新初始化；在对象被销毁后其他方式引用该对象的结果是未定义的。

`pthread_mutexattr_init()` 函数应使用实现定义的所有属性的默认值初始化互斥锁属性对象 `attr`。

如果调用 `pthread_mutexattr_init()` 时指定已经初始化的 `attr` 属性对象，则结果未定义。

在互斥锁属性对象已用于初始化一个或多个互斥锁后，任何影响该属性对象的函数（包括销毁）都不应影响任何先前已初始化的互斥锁。

如果传递给 `pthread_mutexattr_destroy()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，则行为未定义。

返回值

成功完成，`pthread_mutexattr_destroy()` 和 `pthread_mutexattr_init()` 应返回零；否则，应返回错误码以指示错误。

错误

`pthread_mutexattr_destroy()` 函数可能失败的情况：

- **EINVAL** - `attr` 指定的值不引用已初始化的互斥锁属性对象。

`pthread_mutexattr_init()` 函数可能失败的情况：

- **ENOMEM** - 内存不足，无法初始化互斥锁属性对象。

这些函数不应返回 [EINTR] 错误码。

示例

无。

应用用法

如果实现检测到传递给 `pthread_mutexattr_destroy()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，建议函数应失败并报告 [EINVAL] 错误。

有关属性的一般说明，请参见 `pthread_attr_destroy()`。属性对象允许实现尝试有用的扩展，并允许扩展 POSIX.1-2024 而不更改现有函数。因此，它们提供了 POSIX.1-2024 的未来可扩展性，并减少了过早标准化的诱惑，那些语义尚未被广泛实现或理解。

已讨论的可能附加互斥锁属性的示例包括 `spin_only`、`limited_spin`、`no_spin`、`recursive` 和 `metered`。（解释后两种属性可能意味着什么：递归互斥锁将允许当前所有者多次重新锁定；计量互斥锁将透明地保留队列长度、等待时间等记录。）由于基于共享实现和使用经验对这些属性的有用性尚未达成广泛共识，因此它们尚未在 POSIX.1-2024 中指定。但是，互斥锁属性对象使得可以测试这些概念，以便在以后可能进行标准化。

互斥锁属性与性能

已谨慎确保互斥锁属性的默认值被正确定义，使得使用默认值初始化的互斥锁具有足够简单的语义，以便锁定和解锁可以使用等效的测试并设置指令（加上可能的一些其他基本指令）来完成。

如果互斥锁具有非默认属性，至少有一种实现方法可用于减少锁定时测试的成本。实现可以使用的一种方法（并且这可以对完全符合 POSIX 的应用程序完全透明）是秘密预锁定任何初始化为非默认属性的互斥锁。任何稍后尝试锁定此类互斥锁的操作都会导致实现分支到“慢速路径”，就像互斥锁不可用一样；然后，在慢速路径上，实现可以执行“真正的工作”来锁定非默认互斥锁。底层的解锁操作更复杂，因为实现永远不想真正释放这种类型互斥锁的预锁定。这表明，根据硬件的不同，可能存在某些优化，使得无论什么互斥锁属性被认为是“最常用”的，都可以最有效地处理。

进程共享内存与同步

POSIX.1-2024 中内存映射函数的存在使得应用程序可能在多个进程访问的内存中分配本节的同步对象（因此，由多个进程的线程访问）。

为了允许这种使用，同时保持通常情况（即在单个进程中使用）高效，定义了 `process-shared` 选项。

如果实现支持 `_POSIX_THREAD_PROCESS_SHARED` 选项，那么 `process-shared` 属性可用于指示互斥锁或条件变量可能被多个进程的线程访问。

为 `process-shared` 属性选择 `PTHREAD_PROCESS_PRIVATE` 默认设置，以便默认创建这些同步对象的最有效形式。

使用 `PTHREAD_PROCESS_PRIVATE` `process-shared` 属性初始化的同步变量只能由初始化它们的进程中的线程操作。使用 `PTHREAD_PROCESS_SHARED` `process-shared` 属性初始化的同步变量可以由任何有权访问它的进程中的任何线程操作。特别是，这些进程可能存在于初始化进程的生命周期之外。例如，以下代码在映射文件中实现了一个简单的计数信号量，可被许多进程使用。

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};

typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
void semaphore_post(semaphore_t *semaphore);
void semaphore_wait(semaphore_t *semaphore);
void semaphore_close(semaphore_t *semaphore);
```

```
/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
{
    int fd;
```

```
semaphore_t *semap;
pthread_mutexattr_t psharedm;
pthread_condattr_t psharedc;

fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
if (fd < 0)
    return (NULL);
(void) ftruncate(fd, sizeof(semaphore_t));
(void) pthread_mutexattr_init(&psharedm);
(void) pthread_mutexattr_setpshared(&psharedm,
    PTHREAD_PROCESS_SHARED);
(void) pthread_condattr_init(&psharedc);
(void) pthread_condattr_setpshared(&psharedc,
    PTHREAD_PROCESS_SHARED);
semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
    PROT_READ | PROT_WRITE, MAP_SHARED,
    fd, 0);
close (fd);
(void) pthread_mutex_init(&semap->lock, &psharedm);
(void) pthread_cond_init(&semap->nonzero, &psharedc);
semap->count = 0;
return (semap);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;

    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    return (semap);
}

void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semap->nonzero);
    semap->count++;
    pthread_mutex_unlock(&semap->lock);
}
```

```

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}

```

以下代码用于三个独立的进程，它们在文件 **/tmp/semaphore** 中创建、发送和等待信号量。一旦文件被创建，发送和等待程序即使没有初始化信号量，也会递增和递减计数信号量（根据需要进行等待和唤醒）。

```

/* create.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}

```

```

/* post.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)

```

```
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}
```

```
/* wait.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

原理

无。

未来方向

无。

参见

[pthread_attr_destroy\(\)](#) , [pthread_mutex_destroy\(\)](#) ,
[pthread_mutex_init\(\)](#)

POSIX.1-2024 基本定义卷, [<pthread.h>](#)

变更历史

首次发布于 Issue 5。从 Issue 6 开始包含。

衍生自 IEEE POSIX Pthreads Standard 1003.1c-1995 和 POSIX Threads Extension (1003.1j-2000)。

信息性文本结束。

1.159. pthread_mutexattr_getprioceiling, pthread_mutexattr_setprioceiling

SYNOPSIS (概要)

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t
                                      int *restrict prioceiling);

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                      int prioceiling);
```

DESCRIPTION (描述)

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数分别用于获取和设置由 `attr` 指向的互斥锁属性对象的优先级上限属性，该对象先前由 `pthread_mutexattr_init()` 函数创建。

`prioceiling` 属性包含已初始化互斥锁的优先级上限。`prioceiling` 的值在 SCHED_FIFO 定义的最大优先级范围内。

`prioceiling` 属性定义了已初始化互斥锁的优先级上限，这是受互斥锁保护的临界区执行的最低优先级级别。为避免优先级反转 (priority inversion)，互斥锁的优先级上限应设置为高于或等于可能锁定该互斥锁的所有线程的最高优先级。`prioceiling` 的值在 SCHED_FIFO 调度策略下定义的最大优先级范围内。

如果传递给 `pthread_mutexattr_getprioceiling()` 或 `pthread_mutexattr_setprioceiling()` 的 `attr` 参数值不指向已初始化的互斥锁属性对象，则行为未定义。

RETURN VALUE (返回值)

成功完成时，`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS (错误)

这些函数可能失败，如果：

- [EINVAL]

`prioceiling` 指定的值无效。

- [EPERM]

调用者没有执行该操作的权限。

这些函数不应返回 [EINTR] 错误码。

以下部分为提供信息之用。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

无。

RATIONALE (基本原理)

如果实现检测到传递给 `pthread_mutexattr_getprioceiling()` 或 `pthread_mutexattr_setprioceiling()` 的 `attr` 参数值不指向已初始化的互斥锁属性对象，建议函数应失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (另见)

- `pthread_cond_destroy()`
- `pthread_create()`
- `pthread_mutex_destroy()`
- `<pthread.h>`

CHANGE HISTORY (变更历史)

首次在 Issue 5 中发布。为与 POSIX 线程扩展对齐而包含。

标记为实时线程特性组的一部分。

Issue 6

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数被标记为线程和线程优先级保护选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持线程优先级保护选项，不需要提供存根 (stubs)。

[ENOTSUP] 错误条件已被移除，因为这些函数没有 `protocol` 参数。

为 与 ISO/IEC 9899:1999 标 准 对 齐 ， 向 `pthread_mutexattr_getprioceiling()` 原型添加了 `restrict` 关键字。

Issue 7

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数从线程选项移动到需要支持健壮互斥锁优先级保护选项或非健壮互斥锁优先级保护选项。

对于未初始化互斥锁属性对象的 [EINVAL] 错误被移除；此条件导致未定义行为。

1.160. pthread_mutexattr_getprotocol

概要

```
#include <pthread.h>

int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *re
                                  int *restrict protocol);

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr,
                                  int protocol);
```

描述

`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数分别用于获取和设置由 `attr` 指向的互斥锁属性对象的协议属性，该对象先前由 `pthread_mutexattr_init()` 函数创建。

`protocol` 属性定义了在使用互斥锁时要遵循的协议。`protocol` 的值可以是以下之一：

- `PTHREAD_PRIO_INHERIT` (优先级继承)
- `PTHREAD_PRIO_NONE` (无优先级)
- `PTHREAD_PRIO_PROTECT` (优先级保护)

这些值在 `<pthread.h>` 头文件中定义。该属性的默认值应为 `PTHREAD_PRIO_NONE`。

协议行为

当一个线程拥有具有 `PTHREAD_PRIO_NONE` 协议属性的互斥锁时，其优先级和调度不应受到其互斥锁所有权的影响。

`PTHREAD_PRIO_INHERIT`

当一个线程因为拥有一个或多个具有 `PTHREAD_PRIO_INHERIT` 协议属性的互斥锁而阻塞较高优先级线程时，它应该在其优先级与等待该线程拥有的任何用此协议初始化的互斥锁的最高优先级线程的优先级中的较高者上执行。

当一个线程调用 `pthread_mutex_lock()` 时，互斥锁用具有值 PTHREAD_PRIO_INHERIT 的协议属性初始化，当调用线程因为互斥锁被另一个线程拥有而被阻塞时，该拥有线程应继承调用线程的优先级级别，只要它继续拥有该互斥锁。实现应将其执行优先级更新为其分配的优先级和所有其继承的优先级的最大值。此外，如果这个拥有线程本身在另一个具有值 PTHREAD_PRIO_INHERIT 的协议属性的互斥锁上被阻塞，同样的优先级继承效应以递归方式传播到另一个拥有线程。

PTHREAD_PRIO_PROTECT

当一个线程拥有一个或多个用 PTHREAD_PRIO_PROTECT 协议初始化的互斥锁时，它应该在其优先级与该线程拥有的所有用此属性初始化的互斥锁的最高优先级上限中的较高者上执行，无论是否有其他线程被这些互斥锁中的任何一个阻塞。

多个互斥锁

如果一个线程同时拥有几个用不同协议初始化的互斥锁，它应该在这些协议各自会获得的优先级中的最高优先级上执行。

返回值

成功完成后，`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_mutexattr_setprotocol()` 函数在以下情况下应失败：

- **ENOTSUP** - 由 `protocol` 指定的值是不支持的值。

`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数在以下情况下可能失败：

- **EINVAL** - 由 `attr` 或 `protocol` 指定的值无效。

示例

规范中未提供示例。

应用程序使用

`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数是线程选项以及线程优先级保护或线程优先级继承选项的一部分。

这些函数需要支持非健壮互斥锁优先级保护选项、非健壮互斥锁优先级继承选项、健壮互斥锁优先级保护选项或健壮互斥锁优先级继承选项中的至少一个。

基本原理

协议属性允许应用程序控制互斥锁的优先级行为，以防止优先级反转场景，其中较低优先级的线程持有较高优先级线程所需的互斥锁。

默认值

协议属性的默认值是 PTHREAD_PRIO_NONE，这意味着互斥锁所有权不影响线程优先级或调度。

优先级继承

优先级继承 (PTHREAD_PRIO_INHERIT) 在较高优先级线程等待互斥锁时临时提升持有互斥锁的线程的优先级。这有助于防止无界的优先级反转。

优先级保护

优先级保护 (PTHREAD_PRIO_PROTECT) 为每个互斥锁设置优先级上限。拥有此类互斥锁的线程在其自身优先级或其拥有的所有用此协议初始化的互斥锁的最高优先级上限中的较高者上执行。

未来方向

无。

另请参阅

- `pthread_mutexattr_init()`
- `pthread_mutex_lock()`

- `<pthread.h>`

版权

本文本的部分内容摘录并转载自 IEEE Std 1003.1-2017、The Open Group 基本规范第 7 版的电子形式，版权归电气和电子工程师协会（Institute of Electrical and Electronics Engineers, Inc）和 The Open Group 所有，版权所有 © 2017。如果本版本与原始 IEEE 和 The Open Group 标准之间存在任何差异，应以原始 IEEE 和 The Open Group 标准为准。原始标准可在线获取：<http://www.opengroup.org/unix/online.html>。

本页中出现的任何排版或格式错误很可能是在源文件转换为 man 页面格式过程中引入的。要报告此类错误，请参见 https://www.kernel.org/doc/man-pages/reporting_bugs.html。

1.161. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` — 获取和设置互斥锁类 型属性

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict
                               int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int ty
```

DESCRIPTION

`pthread_mutexattr_gettype()` 和 `pthread_mutexattr_settype()` 函数分别用于获取和设置互斥锁的 `type` 属性。该属性通过这些函数的 `type` 参数设置。`type` 属性的默认值为 `PTHREAD_MUTEX_DEFAULT`。

互斥锁的类型包含在互斥锁属性的 `type` 属性中。有效的互斥锁类型包括：

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

互斥锁类型影响加锁和解锁互斥锁的调用行为。详细信息请参见 `pthread_mutex_lock()`。实现可以将 `PTHREAD_MUTEX_DEFAULT` 映射到其他互斥锁类型之一。

如果传递给 `pthread_mutexattr_gettype()` 或 `pthread_mutexattr_settype()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象，则行为未定义。

RETURN VALUE

成功完成后, `pthread_mutexattr_gettype()` 函数应返回零, 并将 `attr` 的 `type` 属性值存储到 `type` 参数引用的对象中。否则, 应返回错误以指示错误。

如果成功, `pthread_mutexattr_settype()` 函数应返回零; 否则, 应返回错误号以指示错误。

ERRORS

`pthread_mutexattr_settype()` 函数在以下情况下将失败:

[EINVAL]

`type` 值无效。

这些函数不应返回 `[EINTR]` 错误代码。

以下部分为参考信息。

EXAMPLES

无。

APPLICATION USAGE

建议应用程序不要将 `PTHREAD_MUTEX_RECURSIVE` 互斥锁与条件变量一起使用, 因为在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中执行的隐式解锁可能不会真正释放互斥锁 (如果它已被多次锁定)。如果发生这种情况, 没有其他线程可以满足谓词的条件。

RATIONALE

如果实现检测到传递给 `pthread_mutexattr_gettype()` 或 `pthread_mutexattr_settype()` 的 `attr` 参数值不引用已初始化的互斥锁属性对象, 建议函数应失败并报告 `[EINVAL]` 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_mutex_lock()`
- `<pthread.h>`

CHANGE HISTORY

首次发布于 Issue 5。

Issue 6

应用了 The Open Group Corrigendum U033/3。
`pthread_mutexattr_gettype()` 的 SYNOPSIS 被更新，使第一个参数为 `const pthread_mutexattr_t *` 类型。

为与 ISO/IEC 9899:1999 标准保持一致，在 `pthread_mutexattr_gettype()` 原型中添加了 `restrict` 关键字。

Issue 7

`pthread_mutexattr_gettype()` 和 `pthread_mutexattr_settype()` 函数从 XSI 选项移至 Base。

删除了未初始化互斥锁属性对象的 `[EINVAL]` 错误；此条件会导致未定义行为。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121]。

Issue 8

应用了 Austin Group Defect 1216，添加了 `pthread_cond_clockwait()`。

1.162. `pthread_mutexattr_init`, `pthread_mutexattr_destroy` - 销毁和初始化互斥锁属性对象

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

DESCRIPTION

`pthread_mutexattr_destroy()` 函数应销毁一个互斥锁属性对象；该对象实际上变为未初始化状态。实现可以使 `pthread_mutexattr_destroy()` 将 `attr` 引用的对象设置为无效值。

已销毁的 `attr` 属性对象可以使用 `pthread_mutexattr_init()` 重新初始化；在对象被销毁后以其他方式引用该对象的结果是未定义的。

`pthread_mutexattr_init()` 函数应使用实现定义的所有属性的默认值来初始化互斥锁属性对象 `attr`。

如果在调用 `pthread_mutexattr_init()` 时指定了一个已经初始化的 `attr` 属性对象，则结果是未定义的。

在互斥锁属性对象已被用于初始化一个或多个互斥锁后，任何影响属性对象的函数（包括销毁）都不应影响任何先前初始化的互斥锁。

如果 `pthread_mutexattr_destroy()` 的 `attr` 参数指定的值不引用已初始化的互斥锁属性对象，则行为是未定义的。

RETURN VALUE

成功完成后，这些函数应返回零；否则，应返回错误码以指示错误。

ERRORS

这些函数可能在以下情况下失败：

- `EINVAL` - `attr` 指定的值不引用已初始化的互斥锁属性对象。

这些函数不应返回 `EINTR` 错误码。

EXAMPLES

无。

APPLICATION USAGE

如果实现检测到 `pthread_mutexattr_destroy()` 的 `attr` 参数指定的值不引用已初始化的互斥锁属性对象，建议函数失败并报告 `[EINVAL]` 错误。

有关属性的一般说明，请参见 `pthread_attr_destroy()`。属性对象允许实现尝试有用的扩展，并允许在不更改现有函数的情况下扩展 POSIX.1-2024 的这一卷。因此，它们为 POSIX.1-2024 这一卷的未来可扩展性提供了支持，并减少了过早标准化尚未广泛实现或理解的语义的诱惑。

已讨论的可能附加互斥锁属性示例包括 `spin_only`、`limited_spin`、`no_spin`、`recursive` 和 `metered`。（解释后两种属性可能意味着：递归互斥锁将允许当前所有者多次重新加锁；计量互斥锁将透明地保持队列长度、等待时间等记录。）由于基于共享实现和使用经验对这些属性的用处尚未达成广泛一致，它们尚未在 POSIX.1-2024 这一卷中指定。但是，互斥锁属性对象使得可以测试这些概念，以便在以后的时间进行可能的标准化。

互斥锁属性与性能

已经谨慎确保互斥锁属性的默认值已定义，使得使用默认值初始化的互斥锁具有足够简单的语义，以便加锁和解锁可以使用测试并设置指令（加上可能的一些其他基本指令）的等效方式完成。

如果互斥锁具有非默认属性，至少有一种实现方法可用于减少锁定时测试的成本。实现可以使用的一种此类方法（并且这对于完全符合 POSIX 的应用程序可以是完全透明的）是秘密地预锁定任何初始化为非默认属性的互斥锁。任何稍后尝试锁定此类互斥锁的操作都会导致实现分支到“慢速路径”，就好像互斥锁不可用一样；然后，在慢速路径上，实现可以执行“真正的工作”来锁定非默认互斥锁。底层解锁操作更复杂，因为实现从不真正想要释放此类互斥锁上的预锁。

定。这说明，根据硬件的不同，可能存在某些优化可用于以最高效率处理被视为“最常使用”的任何互斥锁属性。

进程共享内存与同步

POSIX.1-2024 这一卷中内存映射函数的存在导致应用程序可能可以在多个进程（因此，多个进程的线程）访问的内存中分配本节的同步对象。

为了允许此类使用，同时保持通常情况（即，在单个进程内使用）高效，已定义了**进程共享**选项。

如果实现支持 `_POSIX_THREAD_PROCESS_SHARED` 选项，则**进程共享**属性可用于指示互斥锁或条件变量可由多个进程的线程访问。

已为**进程共享**属性选择 `PTHREAD_PROCESS_PRIVATE` 默认设置，以便默认创建这些同步对象的最有效形式。

使用 `PTHREAD_PROCESS_PRIVATE` **进程共享**属性初始化的同步变量只能由初始化它们的进程中的线程操作。使用 `PTHREAD_PROCESS_SHARED` **进程共享**属性初始化的同步变量可由有权访问它的任何进程中的任何线程操作。特别是，这些进程可能在初始化进程的生命周期之外存在。例如，以下代码在映射文件中实现了一个简单的计数信号量，该信号量可被多个进程使用。

```
/* sem.h */
struct semaphore {
    pthread_mutex_t lock;
    pthread_cond_t nonzero;
    unsigned count;
};

typedef struct semaphore semaphore_t;

semaphore_t *semaphore_create(char *semaphore_name);
semaphore_t *semaphore_open(char *semaphore_name);
void semaphore_post(semaphore_t *semaphore);
void semaphore_wait(semaphore_t *semaphore);
void semaphore_close(semaphore_t *semaphore);

/* sem.c */
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <fcntl.h>
#include <pthread.h>
#include "sem.h"

semaphore_t *
semaphore_create(char *semaphore_name)
```

```

{
    int fd;
    semaphore_t *semap;
    pthread_mutexattr_t psharedm;
    pthread_condattr_t psharedc;

    fd = open(semaphore_name, O_RDWR | O_CREAT | O_EXCL, 0666);
    if (fd < 0)
        return (NULL);
    (void) ftruncate(fd, sizeof(semaphore_t));
    (void) pthread_mutexattr_init(&psharedm);
    (void) pthread_mutexattr_setpshared(&psharedm,
        PTHREAD_PROCESS_SHARED);
    (void) pthread_condattr_init(&psharedc);
    (void) pthread_condattr_setpshared(&psharedc,
        PTHREAD_PROCESS_SHARED);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    (void) pthread_mutex_init(&semap->lock, &psharedm);
    (void) pthread_cond_init(&semap->nonzero, &psharedc);
    semap->count = 0;
    return (semap);
}

semaphore_t *
semaphore_open(char *semaphore_name)
{
    int fd;
    semaphore_t *semap;

    fd = open(semaphore_name, O_RDWR, 0666);
    if (fd < 0)
        return (NULL);
    semap = (semaphore_t *) mmap(NULL, sizeof(semaphore_t),
        PROT_READ | PROT_WRITE, MAP_SHARED,
        fd, 0);
    close (fd);
    return (semap);
}

void
semaphore_post(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    if (semap->count == 0)
        pthread_cond_signal(&semap->nonzero);
    semap->count++;
}

```

```

    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_wait(semaphore_t *semap)
{
    pthread_mutex_lock(&semap->lock);
    while (semap->count == 0)
        pthread_cond_wait(&semap->nonzero, &semap->lock);
    semap->count--;
    pthread_mutex_unlock(&semap->lock);
}

void
semaphore_close(semaphore_t *semap)
{
    munmap((void *) semap, sizeof(semaphore_t));
}

```

以下代码用于三个独立的进程，这些进程在文件 `/tmp/semaphore` 中创建、发布和等待信号量。一旦文件被创建，发布和等待程序就会递增和递减计数信号量（根据需要等待和唤醒），即使它们没有初始化信号量。

```

/* create.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_create("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_close(semap);
    return (0);
}

/* post.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

```

```
    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_post(semap);
    semaphore_close(semap);
    return (0);
}

/* wait.c */
#include "pthread.h"
#include "sem.h"

int
main(void)
{
    semaphore_t *semap;

    semap = semaphore_open("/tmp/semaphore");
    if (semap == NULL)
        exit(1);
    semaphore_wait(semap);
    semaphore_close(semap);
    return (0);
}
```

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [pthread_attr_destroy\(\)](#)
- [pthread_mutex_init\(\)](#)
- [pthread_mutexattr_getpshared\(\)](#)
- [pthread_mutexattr_setpshared\(\)](#)

COPYRIGHT

本文的部分内容转载并电子复制自 IEEE Std 1003.1-2024, 信息技术标准 -- 便携式操作系统接口 (POSIX), The Open Group 基本规范第 7 版, 2024 年版 (IEEE Std 1003.1-2024)。版权所有 © 2024 电气与电子工程师协会和 The Open Group。此版权材料根据 BSD 许可证的条款提供, 该许可证可在 <https://opensource.org/licenses/BSD-3-Clause> 获取。

1.163. `pthread_mutexattr_getprioceiling`, `pthread_mutexattr_setprioceiling`

概要

```
#include <pthread.h>

int pthread_mutexattr_getprioceiling(const pthread_mutexattr_t
                                      int *restrict prioceiling)

int pthread_mutexattr_setprioceiling(pthread_mutexattr_t *attr,
                                      int prioceiling);
```

描述

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数分别用于获取和设置由 `attr` 指向的互斥属性对象的优先级上限属性，该对象先前由 `pthread_mutexattr_init()` 函数创建。

`prioceiling` 属性包含已初始化互斥的优先级上限。`prioceiling` 的值在 SCHED_FIFO 定义的最大优先级范围内。

`prioceiling` 属性定义已初始化互斥的优先级上限，这是执行互斥保护的临界区的最低优先级级别。为避免优先级反转，互斥的优先级上限应设置为高于或等于可能锁定该互斥的所有线程的最高优先级。`prioceiling` 的值在 SCHED_FIFO 调度策略下定义的最大优先级范围内。

如果 `pthread_mutexattr_getprioceiling()` 或 `pthread_mutexattr_setprioceiling()` 的 `attr` 参数指定的值不引用已初始化的互斥属性对象，则行为未定义。

返回值

成功完成后，`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

这些函数可能在以下情况下失败：

- **[EINVAL]**

`prioceiling` 指定的值无效。

- **[EPERM]**

调用者没有执行该操作的权限。

这些函数不应返回 [EINTR] 错误代码。

以下部分为参考信息。

示例

无。

应用程序用法

无。

原理说明

如果实现检测到 `pthread_mutexattr_getprioceiling()` 或 `pthread_mutexattr_setprioceiling()` 的 `attr` 参数指定的值不引用已初始化的互斥属性对象，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

另请参阅

`pthread_cond_destroy()` ,
`pthread_mutex_destroy()`

`pthread_create()` ,

变更历史

首次在 Issue 5 中发布。为与 POSIX 线程扩展对齐而包含。

标记为实时线程特性组的一部分。

Issue 6

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数被标记为线程和线程优先级保护选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持线程优先级保护选项，则不需要提供存根。

[ENOTSUP] 错误条件已被移除，因为这些函数没有 `protocol` 参数。

为与 ISO/IEC 9899:1999 标准对齐，向 `pthread_mutexattr_getprioceiling()` 原型添加了 `restrict` 关键字。

Issue 7

`pthread_mutexattr_getprioceiling()` 和 `pthread_mutexattr_setprioceiling()` 函数从线程选项移动到需要支持健壮互斥优先级保护选项或非健壮互斥优先级保护选项。

针对未初始化互斥属性对象的 [EINVAL] 错误已被移除；此条件导致未定义行为。

1.164. `pthread_mutexattr_setprotocol`, `pthread_mutexattr_getprotocol`

获取和设置互斥锁属性对象的协议属性。

函数概要

```
#include <pthread.h>

int pthread_mutexattr_setprotocol(pthread_mutexattr_t *attr, in
int pthread_mutexattr_getprotocol(const pthread_mutexattr_t *re
```

参数

- `attr` - 指向先前由 `pthread_mutexattr_init()` 函数创建的互斥锁属性对象的指针
- `protocol` - 协议属性的值（对于设置函数）
- `protocol` - 指向存储协议属性的整数的指针（对于获取函数）

返回值

成功时，函数返回 0。失败时，返回错误编号以指示错误。

错误

- `ENOSYS` - 实现不支持指定的协议值
- `ENOTSUP` - 实现不支持指定的协议值
- `EINVAL` - `protocol` 的值无效

描述

`pthread_mutexattr_getprotocol()` 和 `pthread_mutexattr_setprotocol()` 函数分别获取和设置由 `attr` 指向的

互斥锁属性对象的协议属性，该对象先前由 `pthread_mutexattr_init()` 函数创建。

协议属性定义了使用互斥锁时要遵循的协议。`protocol` 的值可以是以下之一：

- `PTHREAD_PRIO_INHERIT` - 优先级继承协议
- `PTHREAD_PRIO_NONE` - 无特殊协议（默认值）
- `PTHREAD_PRIO_PROTECT` - 优先级保护协议

这些值在 `<pthread.h>` 头文件中定义。属性的默认值应为 `PTHREAD_PRIO_NONE`。

协议行为详解

`PTHREAD_PRIO_NONE`

当线程拥有具有 `PTHREAD_PRIO_NONE` 协议属性的互斥锁时，其优先级和调度不应受到其互斥锁所有权的影响。

`PTHREAD_PRIO_INHERIT`

当线程由于拥有一个或多个使用 `PTHREAD_PRIO_INHERIT` 协议属性初始化的互斥锁而阻塞更高优先级的线程时，它应以其优先级或等待该线程拥有的、使用此协议初始化的任何互斥锁的最高优先级线程的优先级中较高的那个优先级执行。

当线程调用 `pthread_mutex_lock()` 时，互斥锁使用 `PTHREAD_PRIO_INHERIT` 协议属性值初始化，当调用线程因互斥锁被另一个线程拥有而被阻塞时，该拥有者线程应继承调用线程的优先级级别，只要它继续拥有该互斥锁。实现应将其执行优先级更新为其分配的优先级和所有继承优先级的最大值。此外，如果此拥有者线程本身在另一个具有 `PTHREAD_PRIO_INHERIT` 协议属性值的互斥锁上被阻塞，相同的优先级继承效果应以递归方式传播到该其他拥有者线程。

`PTHREAD_PRIO_PROTECT`

当线程拥有一个或多个使用 `PTHREAD_PRIO_PROTECT` 协议初始化的互斥锁时，它应以其优先级或该线程拥有的、使用此属性初始化的所有互斥锁的最高优先级上限中较高的那个优先级执行，无论是否有其他线程被这些互斥锁阻塞。

多协议场景

如果线程同时拥有几个使用不同协议初始化的互斥锁，它应以其通过这些协议各自获得的优先级中最高的优先级执行。

注意事项

如果 `pthread_mutexattr_getprotocol()` 或 `pthread_mutexattr_setprotocol()` 的 `attr` 参数指定的值不引用已初始化的互斥锁属性对象，则行为未定义。

一致性

这些函数属于 POSIX.1-2008 标准。

这些函数是线程选项以及线程优先级保护或线程优先级继承选项的一部分。

参见

- `pthread_mutexattr_init()`
- `pthread_mutex_lock()`
- `pthread_mutex_init()`
- `<pthread.h>`

版权

© IEEE 2003, 2023 - All rights reserved.

1.165. `pthread_mutexattr_gettype`, `pthread_mutexattr_settype` - 获取和设置互斥锁类型属性

SYNOPSIS

```
#include <pthread.h>

int pthread_mutexattr_gettype(const pthread_mutexattr_t *restrict
                               int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr, int ty
```

DESCRIPTION

`pthread_mutexattr_gettype()` 和 `pthread_mutexattr_settype()` 函数分别用于获取和设置互斥锁类型属性。该属性在这些函数的 `type` 参数中设置。类型属性的默认值为 `PTHREAD_MUTEX_DEFAULT`。

互斥锁的类型包含在互斥锁属性的 `type` 属性中。有效的互斥锁类型包括：

- `PTHREAD_MUTEX_NORMAL`
- `PTHREAD_MUTEX_ERRORCHECK`
- `PTHREAD_MUTEX_RECURSIVE`
- `PTHREAD_MUTEX_DEFAULT`

互斥锁类型会影响锁定和解锁互斥锁的调用行为。详细信息请参见 `pthread_mutex_lock()`。实现可以将 `PTHREAD_MUTEX_DEFAULT` 映射到其他互斥锁类型之一。

如果传递给 `pthread_mutexattr_gettype()` 或 `pthread_mutexattr_settype()` 的 `attr` 参数值没有引用已初始化的互斥锁属性对象，则行为未定义。

RETURN VALUE

成功完成后，`pthread_mutexattr_gettype()` 函数应返回零，并将 attr 的类型属性值存储到 type 参数引用的对象中。否则，应返回错误以指示错误。

如果成功，`pthread_mutexattr_settype()` 函数应返回零；否则，应返回错误编号以指示错误。

ERRORS

`pthread_mutexattr_settype()` 函数在以下情况应失败：

- **[EINVAL]**

type 值无效。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES

无。

APPLICATION USAGE

建议应用程序不要将 PTHREAD_MUTEX_RECURSIVE 互斥锁与条件变量一起使用，因为在 `pthread_cond_clockwait()`、`pthread_cond_timedwait()` 或 `pthread_cond_wait()` 调用中执行的隐式解锁可能不会实际释放互斥锁（如果它已被多次锁定）。如果发生这种情况，没有其他线程能够满足谓词的条件。

RATIONALE

如果实现检测到传递给 `pthread_mutexattr_gettype()` 或 `pthread_mutexattr_settype()` 的 attr 参数值没有引用已初始化的互斥锁属性对象，建议函数应该失败并报告 [EINVAL] 错误。

FUTURE DIRECTIONS

无。

SEE ALSO

- `pthread_cond_clockwait()`
- `pthread_mutex_lock()`
- `<pthread.h>`

CHANGE HISTORY

首次在 Issue 5 中发布。

Issue 6

应用 Open Group Corrigendum U033/3。`pthread_mutexattr_gettype()` 的 SYNOPSIS 更新为第一个参数是 `const pthread_mutexattr_t *` 类型。

为与 ISO/IEC 9899:1999 标准保持一致，向 `pthread_mutexattr_gettype()` 原型添加了 `restrict` 关键字。

Issue 7

`pthread_mutexattr_gettype()` 和 `pthread_mutexattr_settype()` 函数从 XSI 选项移动到 Base。

删除了未初始化互斥锁属性对象的 [EINVAL] 错误；此条件导致未定义行为。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0464 [121]。

Issue 8

应用 Austin Group Defect 1216，添加 `pthread_cond_clockwait()`。

1.166. `pthread_once` — 动态包初始化

概要

```
#include <pthread.h>

int pthread_once(pthread_once_t *once_control,
                 void (*init_routine)(void));

pthread_once_t once_control = PTHREAD_ONCE_INIT;
```

描述

进程中任何线程对给定 `once_control` 的第一次 `pthread_once()` 调用应该调用不带参数的 `init_routine`。随后使用相同 `once_control` 的 `pthread_once()` 调用不会调用 `init_routine`。从 `pthread_once()` 返回时, `init_routine` 应该已经完成。 `once_control` 参数应该确定相关的初始化例程是否已被调用。

`pthread_once()` 函数不是一个取消点。但是, 如果 `init_routine` 是一个取消点并且被取消, 对 `once_control` 的效果就如同 `pthread_once()` 从未被调用一样。

如果对 `init_routine` 的调用被 `longjmp()` 或 `siglongjmp()` 调用终止, 行为是未定义的。

常量 `PTHREAD_ONCE_INIT` 在 `<pthread.h>` 头文件中定义。

如果 `once_control` 具有自动存储期或未被 `PTHREAD_ONCE_INIT` 初始化, `pthread_once()` 的行为是未定义的。

返回值

成功完成后, `pthread_once()` 应返回零; 否则, 应返回错误编号以指示错误。

错误

`pthread_once()` 函数不应返回 [EINTR] 错误代码。

应用用法

如果 `init_routine` 递归地使用相同的 `once_control` 调用 `pthread_once()`，递归调用不会调用指定的 `init_routine`，因此指定的 `init_routine` 不会完成，因此对 `pthread_once()` 的递归调用不会返回。在 `init_routine` 中使用 `longjmp()` 或 `siglongjmp()` 跳转到 `init_routine` 外部的点会阻止 `init_routine` 返回。

原理

一些 C 库是为动态初始化而设计的。也就是说，库的全局初始化在调用库中的第一个过程时执行。在单线程程序中，这通常使用一个静态变量来实现，在进入例程时检查其值，如下所示：

```
static int random_is_initialized = 0;
extern void initialize_random(void);

int random_function()
{
    if (random_is_initialized == 0) {
        initialize_random();
        random_is_initialized = 1;
    }
    ... /* 初始化后执行的操作。 */
}
```

为了在多线程程序中保持相同的结构，需要一个新的原语。否则，库初始化必须通过在对库的任何使用之前显式调用库导出的初始化函数来完成。

对于多线程进程中的动态库初始化，如果使用初始化标志，则需要保护该标志以防止多个线程同时调用库时对其进行修改。这可以通过使用互斥锁（通过赋值 `PTHREAD_MUTEX_INITIALIZER` 初始化）来完成。但是，更好的解决方案是使用 `pthread_once()`，它正是为此目的而设计的，如下所示：

```
#include <pthread.h>
static pthread_once_t random_is_initialized = PTHREAD_ONCE_INIT
extern void initialize_random(void);

int random_function()
{
    (void) pthread_once(&random_is_initialized, initialize_random);
    ... /* 初始化后执行的操作。 */
}
```

如果实现检测到传递给 `pthread_once()` 的 `once_control` 参数值不引用由 `PTHREAD_ONCE_INIT` 初始化的 `pthread_once_t` 对象，建议函数应该失败并报告 [EINVAL] 错误。

未来方向

无。

参见

- `<pthread.h>`

变更历史

首次发布于 Issue 5。包含用于与 POSIX 线程扩展对齐。

Issue 6

- `pthread_once()` 函数被标记为线程选项的一部分。
- 添加了 [EINVAL] 错误作为"可能失败"情况，如果任一参数无效。

Issue 7

- `pthread_once()` 函数从线程选项移动到基础。
- 移除了未初始化 `pthread_once_t` 对象的 [EINVAL] 错误；此条件导致未定义行为。
- 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0284 [863], XSH/TC2-2008/0285 [874], XSH/TC2-2008/0286 [874]，和 XSH/TC2-2008/0287 [747]。

Issue 8

- 应用了 Austin Group Defect 1330，移除了过时的接口。

1.167. pthread_self — 获取调用线程ID

概要

```
#include <pthread.h>  
  
pthread_t pthread_self(void);
```

描述

`pthread_self()` 函数应返回调用线程的线程ID。

返回值

`pthread_self()` 函数应总是成功的，没有保留任何返回值来表示错误。

错误

未定义错误。

以下章节为参考信息。

示例

无。

应用用法

无。

基本原理

`pthread_self()` 函数提供了类似于进程的 `getpid()` 函数的功能，其基本原理相同：创建调用不会向被创建的线程提供线程ID。

未来方向

无。

参见

- `pthread_create()`
- `pthread_equal()`
- `<pthread.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 线程扩展对齐而包含在内。

Issue 6

`pthread_self()` 函数被标记为 Threads (线程) 选项的一部分。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #063，更新了返回值部分。

`pthread_self()` 函数从 Threads (线程) 选项移动到 Base (基础)。

1.168. pthread_setcancelstate, pthread_setcanceltype, pthread_testcancel

SYNOPSIS

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

DESCRIPTION

`pthread_setcancelstate()` 函数应该原子性地将调用线程的可取消性状态设置为指定的 `state`，并在 `oldstate` 引用的位置返回之前的可取消性状态。`state` 的合法值为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数应该原子性地将调用线程的可取消性类型设置为指定的 `type`，并在 `oldtype` 引用的位置返回之前的可取消性类型。`type` 的合法值为 `PTHREAD_CANCEL_DEFERRED` 和 `PTHREAD_CANCEL_ASYNCHRONOUS`。

任何新创建的线程（包括首次调用 `main()` 的线程）的可取消性状态和类型应该分别为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DEFERRED`。

`pthread_testcancel()` 函数应该在调用线程中创建一个取消点。如果可取消性被禁用，`pthread_testcancel()` 函数应该没有效果。

`pthread_setcancelstate()` 函数应该是异步信号安全的。

RETURN VALUE

如果成功，`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数应该返回零；否则，应该返回错误编号以指示错误。

ERRORS

`pthread_setcancelstate()` 函数可能在以下情况失败：

- [EINVAL]
- 指定的状态不是 `PTHREAD_CANCEL_ENABLE` 或 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数可能在以下情况失败：

- [EINVAL]
- 指定的类型不是 `PTHREAD_CANCEL_DEFERRED` 或 `PTHREAD_CANCEL_ASYNCHRONOUS`。

这些函数不应该返回错误代码 [EINVAL]。

EXAMPLES

无。

APPLICATION USAGE

为了编写可以在可取消线程中安全运行的异步信号处理器，必须使用 `pthread_setcancelstate()` 在信号处理器执行的任何可能是取消点的调用期间禁用取消。然而，标准的早期版本不允许严格符合的应用程序从信号处理器调用 `pthread_setcancelstate()`，因为它不被要求是异步信号安全的。在 `pthread_setcancelstate()` 不是异步信号安全的非符合实现中，替代方案是要么确保在执行不是异步取消安全的函数期间阻塞相应的信号，要么在可能传递这些信号时禁用取消。

RATIONALE

`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数控制线程可能被异步取消的时机。为了使取消控制能够以模块化方式使用，需要遵循一些规则。

对象可以被视为过程的泛化。它是作为一个单元编写的一组过程和全局变量，被对象不知道的客户端调用。对象可能依赖于其他对象。

首先，可取消性应该只在进入对象时禁用，永远不应该显式启用。在退出对象时，可取消性状态应该始终恢复到进入对象时的值。

这遵循模块化参数：如果对象的客户端（或使用该对象的对对象的客户端）禁用了可取消性，这是因为客户端不希望在线程在执行某些操作序列时被取消而担心清理工作。如果在这样的状态下调用对象并且它启用了可取消性，并且该线程有待处理的取消请求，那么线程被取消，这与禁用它的客户端的愿望相反。

其次，在进入对象时，可取消性类型可以显式设置为 [延迟](#) 或 [异步](#)。但与可取消性状态一样，在退出对象时，可取消性类型应该始终恢复到进入对象时的值。

最后，只能从异步可取消的线程中调用取消安全的函数。

FUTURE DIRECTIONS

无。

SEE ALSO

- [pthread_cancel\(\)](#)
- XBD [<pthread.h>](#)

CHANGE HISTORY

首次发布于 Issue 5

为了与 POSIX 线程扩展保持一致而包含。

Issue 6

[pthread_setcancelstate\(\)](#) 、 [pthread_setcanceltype\(\)](#) 和 [pthread_testcancel\(\)](#) 函数被标记为线程选项的一部分。

Issue 7

[pthread_setcancelstate\(\)](#) 、 [pthread_setcanceltype\(\)](#) 和 [pthread_testcancel\(\)](#) 函数从线程选项移动到基础。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0294 [622] 和 XSH/TC2-2008/0295 [615]。

Issue 8

应用了 Austin Group Defect 841，要求 `pthread_setcancelstate()` 是异步信号安全的。

1.169. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel` — 设置 可取消状态

概要

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

描述

`pthread_setcancelstate()` 函数应原子性地将调用线程的可取消状态设置为指定的 `state`，并在 `oldstate` 引用的位置返回之前的可取消状态。`state` 的合法值为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数应原子性地将调用线程的可取消类型设置为指定的 `type`，并在 `oldtype` 引用的位置返回之前的可取消类型。`type` 的合法值为 `PTHREAD_CANCEL_DEFERRED` 和 `PTHREAD_CANCEL_ASYNCHRONOUS`。

任何新创建线程（包括首次调用 `main()` 的线程）的可取消状态和类型应分别为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DEFERRED`。

`pthread_testcancel()` 函数应在调用线程中创建一个取消点。如果取消被禁用，`pthread_testcancel()` 函数应没有效果。

`pthread_setcancelstate()` 函数应为异步信号安全。

返回值

如果成功，`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数应返回零；否则，应返回错误编号以指示错误。

错误

`pthread_setcancelstate()` 函数可能在以下情况下失败：

- [EINVAL]
- 指定的状态不是 `PTHREAD_CANCEL_ENABLE` 或 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数可能在以下情况下失败：

- [EINVAL]
- 指定的类型不是 `PTHREAD_CANCEL_DEFERRED` 或 `PTHREAD_CANCEL_ASYNCHRONOUS`。

这些函数不应返回 [EINVAL] 错误码。

示例

无。

应用程序用法

要为可以在可取消线程中安全运行的异步信号编写信号处理程序，必须使用 `pthread_setcancelstate()` 在信号处理程序执行的任何取消点调用期间禁用取消。但是，标准的早期版本不允许严格符合的应用程序从信号处理程序调用 `pthread_setcancelstate()`，因为它不要求是异步信号安全的。在 `pthread_setcancelstate()` 不是异步信号安全的不符合规范的实现上，替代方案是确保在执行不是异步取消安全的函数期间阻塞相应的信号，或者在可能传递这些信号时禁用取消。

基本原理

`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数控制线程可能被异步取消的点。为了使取消控制能够以模块化方式使用，需要遵循一些规则。

对象可以看作是过程的泛化。它是作为单元编写的一组过程和全局变量，由对象不知道的客户端调用。对象可能依赖于其他对象。

首先，取消只应在进入对象时禁用，绝不应显式启用。退出对象时，应始终将取消状态恢复到进入对象时的值。

这是从模块化论证得出的：如果一个对象的客户端（或使用该对象的对象的客户端）禁用了取消，这是因为客户端不关心如果线程在执行某些操作序列时被取消的清理工作。如果在这种状态下调用对象并启用取消，并且该线程有待处理的取消请求，那么线程将被取消，这与禁用的客户端的愿望相违背。

其次，取消类型可以在进入对象时显式设置为延迟或异步。但与取消状态一样，退出对象时应始终将取消类型恢复到进入对象时的值。

最后，只有取消安全的函数才能从可异步取消的线程中调用。

未来方向

无。

另请参阅

- `pthread_cancel()`
- XBD `<pthread.h>`

更改历史

在第5期中首次发布

为了与 POSIX 线程扩展对齐而包含。

第6期

`pthread_setcancelstate()`、`pthread_setcanceltype()` 和 `pthread_testcancel()` 函数被标记为线程选项的一部分。

第7期

`pthread_setcancelstate()`、`pthread_setcanceltype()` 和 `pthread_testcancel()` 函数从线程选项移动到基本规范。

应用了 POSIX.1-2008 技术勘误 2，XSH/TC2-2008/0294 [622] 和 XSH/TC2-2008/0295 [615]。

第8期

应用了 Austin Group 缺陷 841，要求 `pthread_setcancelstate()` 为异步信号安全。

1.170. `pthread_setconcurrency`, `pthread_getconcurrency`

SYNOPSIS

```
#include <pthread.h>

int pthread_setconcurrency(int new_level);
int pthread_getconcurrency(void);
```

DESCRIPTION

进程中的非绑定线程可能需要也可能不需要同时激活。默认情况下，线程实现确保有足够的线程处于活动状态，以便进程能够继续取得进展。虽然这样做节省了系统资源，但可能不会产生最有效的并发级别。

`pthread_setconcurrency()` 函数允许应用程序通知线程实现其期望的并发级别 `new_level`。由于此函数调用，实现提供的实际并发级别是未指定的。

如果 `new_level` 为零，将导致实现自行决定维护并发级别，就像从未调用过 `pthread_setconcurrency()` 一样。

`pthread_getconcurrency()` 函数应返回先前调用 `pthread_setconcurrency()` 函数所设置的值。如果先前未调用过 `pthread_setconcurrency()` 函数，则此函数应返回零，以表示实现正在维护并发级别。

调用 `pthread_setconcurrency()` 应通知实现其期望的并发级别。实现应将此作为提示 (hint)，而非要求。

如果实现不支持在多个内核调度实体之上对用户线程进行多路复用，则提供 `pthread_setconcurrency()` 和 `pthread_getconcurrency()` 函数是为了源代码兼容性，但在调用时它们不起作用。为了保持函数语义，在调用 `pthread_setconcurrency()` 时会保存 `new_level` 参数，以便后续调用 `pthread_getconcurrency()` 能够返回相同的值。

RETURN VALUE

如果成功，`pthread_setconcurrency()` 函数应返回零；否则，应返回错误码以指示错误。

`pthread_getconcurrency()` 函数应始终返回先前调用 `pthread_setconcurrency()` 所设置的并发级别。如果从未调用过 `pthread_setconcurrency()` 函数，`pthread_getconcurrency()` 应返回零。

ERRORS

未定义这些函数的错误。

EXAMPLES

未提供示例。

APPLICATION USAGE

这些函数允许应用程序向线程实现提供有关期望并发级别的提示。实现不要求必须遵循这些提示，实际并发级别可能与请求的级别不同。

RATIONALE

并发级别函数为应用程序提供了一种机制，当它们具有关于同时活动线程最优数量的知识时，可以影响线程调度行为。然而，实现在如何响应这些提示方面被赋予了灵活性，以适应不同的线程模型和系统架构。

FUTURE DIRECTIONS

无。

SEE ALSO

无。

COPYRIGHT

本文本部分内容转载并复制自 IEEE Std 1003.1, 2013 Edition, Standard for Information Technology -- Portable Operating System Interface (POSIX), The Open Group Base Specifications Issue 7, 版权所有 (C) 2013 by the Institute of Electrical and Electronics Engineers, Inc and The Open Group。

(这是应用了 2013 年技术勘误 1 的 POSIX.1-2008。) 如果此版本与原始 IEEE 和 The Open Group 标准之间存在任何差异, 应以原始 IEEE 和 The Open Group 标准为准。原始标准可在线获取 : <http://www.opengroup.org/unix/online.html>。

以下部分为参考信息。

1.171. `pthread_getschedparam`, `pthread_setschedparam`

概要

```
#include <pthread.h>

int pthread_getschedparam(pthread_t thread,
                           int *restrict policy,
                           struct sched_param *restrict param);

int pthread_setschedparam(pthread_t thread,
                           int policy,
                           const struct sched_param *param);
```

描述

`pthread_getschedparam()` 和 `pthread_setschedparam()` 函数分别用于获取和设置多线程进程中单个线程的调度策略和参数。对于 SCHED_FIFO 和 SCHED_RR，`sched_param` 结构中唯一必需的成员是优先级 `sched_priority`。对于 SCHED_OTHER，受影响的调度参数由实现定义。

`pthread_getschedparam()` 函数应获取由 `thread` 给定线程 ID 的线程的调度策略和调度参数，并分别将这些值存储在 `policy` 和 `param` 中。从 `pthread_getschedparam()` 返回的优先级值应是最近一次影响目标线程的 `pthread_setschedparam()`、`pthread_setschedprio()` 或 `pthread_create()` 调用指定的值。它不应反映由于任何优先级继承或优先级上限函数而导致的优先级的任何临时调整。

`pthread_setschedparam()` 函数应为由 `thread` 给定线程 ID 的线程设置调度策略和关联的调度参数，分别设置为 `policy` 和 `param` 中提供的策略和关联参数。

`policy` 参数的值可以是 SCHED_OTHER、SCHED_FIFO 或 SCHED_RR。SCHED_OTHER 策略的调度参数由实现定义。SCHED_FIFO 和 SCHED_RR 策略应具有单一的调度参数 `priority`。

SCHED_SPORADIC 服务器

如果定义了 `_POSIX_THREAD_SPORADIC_SERVER`，则 `policy` 参数的值可以是 `SCHED_SPORADIC`，但对于 `pthread_setschedparam()` 函数存在例外：如果在调用时调度策略不是 `SCHED_SPORADIC`，则是否支持该函数由实现定义；换句话说，实现不需要允许应用程序动态地将调度策略更改为 `SCHED_SPORADIC`。

零星服务器调度策略具有关联参数：

- `sched_ss_low_priority`
- `sched_ss_repl_period`
- `sched_ss_init_budget`
- `sched_priority`
- `sched_ss_max_repl`

为使函数成功，指定的 `sched_ss_repl_period` 必须大于或等于指定的 `sched_ss_init_budget`；如果不是，则函数应失败。`sched_ss_max_repl` 的值必须在包含范围 `[1,{SS_REPL_MAX}]` 内函数才能成功；如果不是，函数应失败。`sched_ss_repl_period` 和 `sched_ss_init_budget` 值是按此函数提供的方式存储还是四舍五入以与所用时钟的分辨率对齐，是未指定的。

如果 `pthread_setschedparam()` 函数失败，目标线程的调度参数不应更改。

返回值

如果成功，`pthread_getschedparam()` 和 `pthread_setschedparam()` 函数应返回零；否则，应返回错误码以指示错误。

错误

`pthread_setschedparam()` 函数在以下情况应失败：

- **ENOTSUP**: 尝试将策略或调度参数设置为不支持的值。
- **ENOTSUP**: 尝试将调度策略动态更改为 `SCHED_SPORADIC`，而实现不支持此更改。

`pthread_setschedparam()` 函数在以下情况可能失败：

- **EINVAL**: 由 `policy` 指定的值或与调度策略 `policy` 关联的某个调度参数无效。

- **EPERM**: 调用者没有适当的权限来设置指定线程的调度参数或调度策略。
- **EPERM**: 实现不允许应用程序将某个参数修改为指定的值。

这些函数不应返回错误码 [\[EINTR\]](#)。

示例

无。

应用程序使用

无。

原理

如果实现在线程生命周期结束后检测到线程 ID 的使用，建议函数应失败并报告 [\[ESRCH\]](#) 错误。

未来方向

无。

参见

- [pthread_setschedprio\(\)](#)
- [sched_getparam\(\)](#)
- [sched_getscheduler\(\)](#)
- XBD [<pthread.h>](#)
- XBD [<sched.h>](#)

变更历史

首次发布于 Issue 5

为与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_getschedparam()` 和 `pthread_setschedparam()` 函数被标记为线程和线程执行调度选项的一部分。
- `[ENOSYS]` 错误条件已被移除，因为如果实现不支持线程执行调度选项，不需要提供存根。
- 应用了 The Open Group 勘误表 U026/2，更正了 `pthread_setschedparam()` 函数的原型，使其第二个参数为 `int` 类型。
- 为与 IEEE Std 1003.1d-1999 对齐而添加了 SCHED_SPORADIC 调度策略。
- 为与 ISO/IEC 9899:1999 标准对齐而向 `pthread_getschedparam()` 原型添加了 `restrict` 关键字。
- 应用了 The Open Group 勘误表 U047/1。
- 应用了 IEEE PASC 解释 1003.1 #96，指出优先级值也可以通过对 `pthread_setschedprio()` 函数的调用来设置。

Issue 7

- `pthread_getschedparam()` 和 `pthread_setschedparam()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。
- 应用了 Austin Group 解释 1003.1-2001 #119，阐明了 `sched_ss_repl_period` 和 `sched_ss_init_budget` 值的精度要求。
- 应用了 Austin Group 解释 1003.1-2001 #142，移除了 `[ESRCH]` 错误条件。
- 应用了 POSIX.1-2008 技术勘误表 1，XSH/TC1-2008/0459 [314]。
- 应用了 POSIX.1-2008 技术勘误表 2，XSH/TC2-2008/0276 [757]。

1.172. pthread_setschedprio

概要

```
#include <pthread.h>

int pthread_setschedprio(pthread_t thread, int prio);
```

描述

`pthread_setschedprio()` 函数应将由 `thread` 给出的线程ID的线程的调度优先级设置为 `prio` 给出的值。关于此函数调用如何影响线程在新优先级下的线程列表中排序的描述，请参见调度策略。

如果 `pthread_setschedprio()` 函数失败，目标线程的调度优先级不应被更改。

返回值

如果成功，`pthread_setschedprio()` 函数应返回零；否则，应返回错误号以指示错误。

错误

`pthread_setschedprio()` 函数可能在以下情况下失败：

- **[EINVAL]**

`prio` 的值对于指定线程的调度策略无效。

- **[EPERM]**

调用者没有适当的权限来设置指定线程的调度优先级。

`pthread_setschedprio()` 函数不应返回 [EINTR] 错误码。

示例

无。

应用程序用法

无。

原理

`pthread_setschedprio()` 函数为应用程序提供了一种暂时提高其优先级然后再次降低优先级的方法，而不会产生让位于相同优先级的其他线程的不良副作用。如果应用程序要实现自己的限制优先级反转的策略，如优先级继承或优先级上限，这是必需的。如果实现不支持线程优先级保护或线程优先级继承选项，这种能力尤其重要，但即使支持这些选项，如果应用程序要为其他资源（如信号量）限制优先级继承，也需要此功能。

标准开发者认为，虽然从概念上讲，通过修改 `pthread_setschedparam()` 的规范来解决这个问题可能更可取，但进行此类更改为时已晚，因为可能需要更改某些实现。因此，引入了这个新函数。

如果实现在线程生命期结束后检测到线程ID的使用，建议该函数应失败并报告 [ESRCH] 错误。

未来方向

无。

另请参见

- 调度策略
- `pthread_getschedparam()`
- `<pthread.h>`

变更历史

首次在 Issue 6 中发布。作为对 IEEE PASC Interpretation 1003.1 #96 的响应而包含。

Issue 7

`pthread_setschedprio()` 函数仅被标记为线程执行调度选项的一部分，因为线程选项现在是基础的一部分。

应用了 Austin Group Interpretation 1003.1-2001 #069，更新了 [EPERM] 错误。

应用了 Austin Group Interpretation 1003.1-2001 #142，移除了 [ESRCH] 错误条件。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0466 [314]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0296 [757]。

1.173. pthread_getspecific, pthread_setspecific — 线程特定数据管理

概要

```
#include <pthread.h>

void *pthread_getspecific(pthread_key_t key);

int pthread_setspecific(pthread_key_t key, const void *value);
```

描述

`pthread_getspecific()` 函数应返回当前绑定到指定 `key` 的值，该值代表调用线程。

`pthread_setspecific()` 函数应将线程特定的 `value` 与通过先前调用 `pthread_key_create()` 获得的 `key` 关联起来。不同的线程可以将不同的值绑定到同一个键。这些值通常是指向动态分配内存块的指针，这些内存块已保留供调用线程使用。

使用未从 `pthread_key_create()` 获得的 `key` 值调用 `pthread_getspecific()` 或 `pthread_setspecific()`，或在 `key` 已被 `pthread_key_delete()` 删除后调用这些函数的效果是未定义的。

`pthread_getspecific()` 和 `pthread_setspecific()` 都可以从线程特定数据析构函数中调用。对正在销毁的线程特定数据键调用 `pthread_getspecific()` 应返回值 `NULL`，除非该值在析构函数开始后通过调用 `pthread_setspecific()` 被更改。从线程特定数据析构例程中调用 `pthread_setspecific()` 可能导致存储丢失（在至少 `PTHREAD_DESTRUCTOR_ITERATIONS` 次销毁尝试后）或无限循环。

这两个函数都可以实现为宏。

返回值

`pthread_getspecific()` 函数应返回与给定 `key` 关联的线程特定数据值。如果没有与 `key` 关联的线程特定数据值，则应返回值 `NULL`。

如果成功，`pthread_setspecific()` 函数应返回零；否则，应返回一个错误号以指示错误。

错误

`pthread_getspecific()` 不返回任何错误。

`pthread_setspecific()` 函数在以下情况下可能失败：

- [ENOMEM]
- 没有足够的内存来将非 NULL 值与键关联。

`pthread_setspecific()` 函数不应返回 [EINTR] 错误码。

示例

无。

应用用法

无。

原理

`pthread_getspecific()` 的性能和易用性对于依赖在线程特定数据中维护状态的函数至关重要。由于它不需要检测任何错误，并且由于唯一可能检测到的错误是无效键的使用，`pthread_getspecific()` 函数的设计优先考虑速度和简单性，而不是错误报告。

如果实现检测到 `pthread_setspecific()` 的 `key` 参数指定的值不是指从 `pthread_key_create()` 获得的键值，或者是已被 `pthread_key_delete()` 删除的键，建议函数应该失败并报告 [EINVAL] 错误。

未来方向

无。

另请参阅

- `pthread_key_create()`
- `<pthread.h>`

变更历史

首次发布于 Issue 5

为了与 POSIX 线程扩展对齐而包含。

Issue 6

- `pthread_getspecific()` 和 `pthread_setspecific()` 函数被标记为线程选项的一部分。
- 应用了 IEEE PASC Interpretation 1003.1c #3 (Part 6)，更新了描述部分。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/96，更新了错误部分，使 [ENOMEM] 错误情况从"将值与键关联"更改为"将非 NULL 值与键关联"。

Issue 7

- 应用了 Austin Group Interpretation 1003.1-2001 #063，更新了错误部分。
- `pthread_getspecific()` 和 `pthread_setspecific()` 函数从线程选项移动到基础部分。
- 删除了对于未从 `pthread_key_create()` 获得的键值或已被 `pthread_key_delete()` 删除的键的 [EINVAL] 错误；这种情况会导致未定义行为。

1.174. `pthread_sigmask`, `sigprocmask` — 检查和更改阻塞信号

概要

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                     sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

描述

`pthread_sigmask()` 函数用于检查、更改（或同时进行两者）调用线程的信号掩码。

如果参数 `set` 不是空指针，它指向一个用于更改当前阻塞信号集合的信号集合。

参数 `how` 指示更改集合的方式，应用程序应确保其值由以下之一组成：

`SIG_BLOCK`

- 结果集合应为当前集合与 `set` 指向的信号集合的并集。

`SIG_SETMASK`

- 结果集合应为 `set` 指向的信号集合。

`SIG_UNBLOCK`

- 结果集合应为当前集合与 `set` 指向的信号集合的补集的交集。

如果参数 `oset` 不是空指针，则应将之前的掩码存储在 `oset` 指向的位置。如果 `set` 是空指针，则参数 `how` 的值没有意义，且线程的信号掩码应保持不变；因此该调用可用于查询当前被阻塞的信号。

如果参数 `set` 不是空指针，在 `pthread_sigmask()` 更改当前阻塞信号集合后，它应确定是否有任何挂起的未阻塞信号；如果有，则在 `pthread_sigmask()` 调用返回之前至少应传递这些信号中的一个。

无法阻塞那些不能被忽略的信号。系统应在不导致错误指示的情况下强制执行此规定。

如果 SIGFPE、SIGILL、SIGSEGV 或 SIGBUS 信号中的任何一个在被阻塞时生成，则结果未定义，除非该信号是由另一个进程的操作，或由 `kill()`、`pthread_kill()`、`raise()` 或 `sigqueue()` 函数之一生成的。

如果 `pthread_sigmask()` 失败，线程的信号掩码不应被更改。

`sigprocmask()` 函数应等同于 `pthread_sigmask()`，不同之处在于如果从多线程进程中调用其行为未指定，并且它在错误时返回 -1 并将 `errno` 设置为错误编号，而不是直接返回错误编号。

返回值

成功完成后，`pthread_sigmask()` 应返回 0；否则，应返回相应的错误编号。

成功完成后，`sigprocmask()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

错误

这些函数在以下情况下可能会失败：

[EINVAL]

- `set` 参数不是空指针，且 `how` 参数的值不等于任何定义的值。

这些函数不应返回 [EINTR] 错误代码。

示例

多线程进程中的信号处理

此示例显示了 `pthread_sigmask()` 在多线程进程中处理信号的使用。它提供了一个相当通用的框架，可以轻松地适配/扩展。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
...
...
```

```
static sigset_t    signal_mask; /* 要阻塞的信号          */

int main (int argc, char *argv[])
{
    pthread_t  sig_thr_id;      /* 信号处理线程 ID */
    int         rc;             /* 返回代码          */

    sigemptyset (&signal_mask);
    sigaddset (&signal_mask, SIGINT);
    sigaddset (&signal_mask, SIGTERM);
    rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
    if (rc != 0) {
        /* 处理错误 */
        ...
    }
    /* 任何新创建的线程都会继承信号掩码 */

    rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL
    if (rc != 0) {
        /* 处理错误 */
        ...
    }
    /* 应用程序代码 */
    ...
}

void *signal_thread (void *arg)
{
    int      sig_caught;      /* 捕获的信号          */
    int      rc;               /* 返回代码          */

    rc = sigwait (&signal_mask, &sig_caught);
    if (rc != 0) {
        /* 处理错误 */
    }
    switch (sig_caught)
    {
    case SIGINT:    /* 处理 SIGINT */
        ...
        break;
    case SIGTERM:   /* 处理 SIGTERM */
        ...
        break;
    default:        /* 通常不应发生 */
        fprintf (stderr, "\n意外信号 %d\n", sig_caught);
        break;
    }
}
```

```
    }  
}
```

应用程序用法

虽然 `pthread_sigmask()` 必须在更改当前阻塞信号集合后传递至少一个存在的挂起未阻塞信号，但没有要求传递的信号包括通过该更改解除阻塞的任何信号。如果在 `pthread_sigmask()` 调用执行期间，一个或多个已经解除阻塞的信号变为挂起状态（参见 2.4.1 信号生成和传递），在调用返回之前传递的信号可能仅包括那些信号。

原理

当在由 `sigaction()` 安装的信号捕获函数中更改线程的信号掩码时，从信号捕获函数返回时信号掩码的恢复会覆盖该更改（参见 `sigaction()`）。如果信号捕获函数是通过 `signal()` 安装的，则是否发生这种情况未指定。

关于信号传递要求，请参见 `kill()` 的讨论。

未来方向

无。

另请参见

`exec` , `kill()` , `sigaction()` , `sigaddset()` , `sigdelset()` ,
`sigemptyset()` , `sigfillset()` , `sigismember()` , `sigpending()` ,
`sigqueue()` , `sigsuspend()`

XBD `<signal.h>`

更改历史

首次在 Issue 3 中发布。为与 POSIX.1-1988 标准对齐而包含。

Issue 5

- 描述已更新，以与 POSIX 线程扩展对齐。
- 为与 POSIX 线程扩展对齐而添加了 `pthread_sigmask()` 函数。

Issue 6

- `pthread_sigmask()` 函数被标记为线程选项的一部分。
- `sigprocmask()` 的概要被标记为 CX 扩展，以注意该函数在 `<signal.h>` 头文件中的存在是对 ISO C 标准的扩展。
- 为与 ISO POSIX-1:1996 标准对齐而进行了以下更改：
- 描述已更新，以明确说明可能生成信号的函数。
- 规范性文本已更新，以避免在应用程序要求中使用术语"必须"。
- 为与 ISO/IEC 9899:1999 标准对齐，`restrict` 关键字被添加到 `pthread_sigmask()` 和 `sigprocmask()` 原型中。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/105，将描述和原理部分中的"进程的信号掩码"更新为"线程的信号掩码"。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/106，向示例部分添加了示例。

Issue 7

- `pthread_sigmask()` 函数从线程选项移动到基础。
- POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0467 [319] 被应用。

Issue 8

- 应用了 Austin Group 缺陷 1132，澄清了 [EINVAL] 错误。
- 应用了 Austin Group 缺陷 1636，澄清了 `pthread_sigmask()` 和 `sigprocmask()` 等价性的例外情况。
- 应用了 Austin Group 缺陷 1731，澄清了虽然 `pthread_sigmask()` 必须在更改当前阻塞信号集合后传递至少一个存在的挂起未阻塞信号，但没有要求传递的信号包括通过该更改解除阻塞的任何信号。

1.175. `pthread_setcancelstate`, `pthread_setcanceltype`, `pthread_testcancel`

概要 (SYNOPSIS)

```
#include <pthread.h>

int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

描述 (DESCRIPTION)

`pthread_setcancelstate()` 函数应原子地设置调用线程的可取消状态为指定的 `state`，并在 `oldstate` 引用的位置返回先前的可取消状态。`state` 的合法值为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数应原子地设置调用线程的可取消类型为指定的 `type`，并在 `oldtype` 引用的位置返回先前的可取消类型。`type` 的合法值为 `PTHREAD_CANCEL_DEFERRED` 和 `PTHREAD_CANCEL_ASYNCHRONOUS`。

任何新创建线程（包括首次调用 `main()` 的线程）的可取消状态和类型应分别为 `PTHREAD_CANCEL_ENABLE` 和 `PTHREAD_CANCEL_DEFERRED`。

`pthread_testcancel()` 函数应在调用线程中创建一个取消点。如果取消被禁用，`pthread_testcancel()` 函数应无效。

`pthread_setcancelstate()` 函数应为异步信号安全函数。

返回值 (RETURN VALUE)

如果成功，`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数应返回零；否则，应返回错误码以指示错误。

错误 (ERRORS)

`pthread_setcancelstate()` 函数可能失败，如果：

- `[EINVAL]`

- 指定的状态不是 `PTHREAD_CANCEL_ENABLE` 或 `PTHREAD_CANCEL_DISABLE`。

`pthread_setcanceltype()` 函数可能失败，如果：

- `[EINVAL]`
- 指定的类型不是 `PTHREAD_CANCEL_DEFERRED` 或 `PTHREAD_CANCEL_ASYNCHRONOUS`。

这些函数不应返回 `[EINTR]` 错误码。

以下章节为参考信息。

示例 (EXAMPLES)

无。

应用程序使用 (APPLICATION USAGE)

为了为可以在可取消线程中安全运行的异步信号编写信号处理程序，必须使用 `pthread_setcancelstate()` 在信号处理程序执行的任何可能作为取消点的调用期间禁用取消。但是，标准的早期版本不允许严格符合的应用程序从信号处理程序调用 `pthread_setcancelstate()`，因为它不要求是异步信号安全的。在 `pthread_setcancelstate()` 不是异步信号安全的不符合实现上，替代方案是确保在执行非异步取消安全函数期间阻塞相应的信号，或者在可能传递这些信号时禁用取消。

原理 (RATIONALE)

`pthread_setcancelstate()` 和 `pthread_setcanceltype()` 函数控制线程可能被异步取消的时间点。为了使取消控制能够以模块化方式使用，需要遵循一些规则。

对象可以被认为是过程的泛化。它是作为单元编写的一组过程和全局变量，并被对象不知道的客户端调用。对象可能依赖于其他对象。

首先，可取消性只能在进入对象时禁用，绝不能显式启用。在退出对象时，可取消状态应始终恢复到进入对象时的值。

这遵循模块化参数：如果对象的客户端（或使用该对象的对象的客户端）禁用了可取消性，这是因为客户端不关心如果线程在执行某些操作序列时被取消时进行清理。如果在这种状态下调用对象并启用了可取消性，并且该线程有待处理的取消请求，那么线程被取消，这与禁用客户端的愿望相反。

其次，可取消类型可以在进入对象时显式设置为延迟或异步。但与可取消状态一样，在退出对象时，可取消类型应始终恢复到进入对象时的值。

最后，只有取消安全的函数才能从可异步取消的线程中调用。

未来方向 (FUTURE DIRECTIONS)

无。

另请参见 (SEE ALSO)

- `pthread_cancel()`
- XBD `<pthread.h>`

更改历史 (CHANGE HISTORY)

首次在 Issue 5 中发布。包含用于与 POSIX 线程扩展对齐。

Issue 6

`pthread_setcancelstate()`、`pthread_setcanceltype()` 和 `pthread_testcancel()` 函数被标记为线程选项的一部分。

Issue 7

`pthread_setcancelstate()`、`pthread_setcanceltype()` 和 `pthread_testcancel()` 函数从线程选项移动到基础。

POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0294 [622] 和 XSH/TC2-2008/0295 [615] 被应用。

Issue 8

Austin Group Defect 841 被应用，要求 `pthread_setcancelstate()` 为异步信号安全。

参考文本结束。

1.176. putc — 向流中写入一个字节

概要

```
#include <stdio.h>

int putc(int c, FILE *stream);
```

描述

`putc()` 函数应等价于 `fputc()`，但如果它被实现为宏，则可能会多次求值 `stream` 参数，因此该参数永远不应该是具有副作用的表达式。

返回值

参考 `fputc()`。

错误

参考 `fputc()`。

示例

无。

应用程序用法

由于 `putc()` 可能被实现为宏，它可能会错误地处理具有副作用的 `stream` 参数。特别是，`putc(c, *f++)` 不一定能正确工作。因此，在这种情况下不推荐使用此函数；应该使用 `fputc()` 代替。

原理

无。

未来方向

无。

参见

2.5 标准 I/O 流, [fputc\(\)](#)

XBD [<stdio.h>](#)

变更历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008 Technical Corrigendum 1, XSH/TC1-2008/0470 [14]。

1.177. putc_unlocked

SYNOPSIS

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

DESCRIPTION

提供的函数版本 `getc_unlocked()`、`getchar_unlocked()`、`putc_unlocked()` 和 `putchar_unlocked()` 分别是 `getc()`、`getchar()`、`putc()` 和 `putchar()` 函数的对应版本，功能上与原始版本等效，但它们不要求以完全线程安全的方式实现。当在受 `flockfile()`（或 `ftrylockfile()`）和 `funlockfile()` 保护的范围内使用时，它们应该是线程安全的。这些函数可以在多线程程序中安全使用，当且仅当在调用线程拥有 `(FILE *)` 对象时调用它们，就像成功调用 `flockfile()` 或 `ftrylockfile()` 函数之后的情况。

如果 `getc_unlocked()` 或 `putc_unlocked()` 作为宏实现，它们可能会多次计算 `stream` 参数，因此 `stream` 参数永远不应该是有副作用的表达式。

RETURN VALUE

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

ERRORS

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

以下部分为补充信息。

EXAMPLES

无。

APPLICATION USAGE

由于它们可能作为宏实现，`getc_unlocked()` 和 `putc_unlocked()` 可能错误地处理有副作用的 `stream` 参数。特别是，`getc_unlocked(f++)` 和 `putc_unlocked(c,f++)` 不一定能按预期工作。因此，在这种情况下使用这些函数之前，应根据需要使用以下语句：

```
#undef getc_unlocked
#undef putc_unlocked
```

RATIONALE

出于性能原因，某些 I/O 函数通常实现为宏（例如，`putc()` 和 `getc()`）。为了安全，它们需要同步，但为每个字符进行同步往往开销过大。尽管如此，仍认为安全问题是更重要的；因此，要求 `getc()`、`getchar()`、`putc()` 和 `putchar()` 函数是线程安全的。然而，也提供了名称中明确指示其操作不安全但可以利用其更高性能的 `unlocked` 版本。这些 `unlocked` 版本只能在使用导出的锁定原语的显式锁定程序区域内安全使用。特别是，如下序列：

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

是允许的，并且输出结果为：

```
1
Line 2
```

不会被其他线程的输出插断。

将 `getc()`、`putc()` 等标准名称映射到“更快但不安全”版本而非“更慢但安全”版本是错误的。无论哪种情况，在转换现有代码时，您仍然希望手动检查 `getc()`、`putc()` 等的所有使用。至少选择安全绑定作为默认，会生成正确的代码并保持“函数级原子性”不变性。否则会在转换后的代码中引入不必要的同步错误。其他修改 `stdio` (`FILE *`) 结构或缓冲区的例程也进行了安全同步。

请注意，不需要 `getc_locked()`、`putc_locked()` 等形式的函数，因为这就是 `getc()`、`putc()` 等的功能。使用特性测试宏在 `getc_locked()` 和 `getc_unlocked()` 之间切换 `getc()` 的宏定义是不合适的，因为 ISO C 标准要求存在实际函数，其行为无法被特性测试宏改变。此外，同时提供 `xxx_locked()` 和 `xxx_unlocked()` 形式会导致混淆，即后缀是描述函数的行为还是应该使用的环境。

提供了三个额外的例程：`flockfile()`、`ftrylockfile()` 和 `funlockfile`（它们可能是宏），允许用户勾勒出同步执行的 I/O 语句序列。

`ungetc()` 函数相对于其他函数/宏的调用频率较低，因此不需要 `unlocked` 变体。

FUTURE DIRECTIONS

无。

SEE ALSO

- [2.5 标准 I/O 流](#)
- [flockfile\(\)](#)
- [getc\(\)](#)
- [getchar\(\)](#)
- [putc\(\)](#)
- [putchar\(\)](#)
- XBD

CHANGE HISTORY

首次在 Issue 5 中发布。为了与 POSIX 线程扩展对齐而包含在内。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

应用了 The Open Group 勘误 U030/2，添加了描述应用程序应如何编写以避免函数作为宏实现的情况的 APPLICATION USAGE 部分。

Issue 7

`getc_unlocked()` 、 `getchar_unlocked()` 、 `putc_unlocked()` 和 `putchar_unlocked()` 函数从线程安全函数选项移动到基础规范。

应用了 POSIX.1-2008 技术勘误 1：XSH/TC1-2008/0232 [395]、XSH/TC1-2008/0233 [395]、XSH/TC1-2008/0234 [395] 和 XSH/TC1-2008/0235 [14]。

应用了 POSIX.1-2008 技术勘误 2：XSH/TC2-2008/0151 [826]。

1.178. putchar

概要

```
#include <stdio.h>

int putchar(int c);
```

描述

[CX] 本参考页所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

函数调用 `putchar(c)` 应等效于 `putc(c, stdout)`。

返回值

参考 `fputc()`。

错误

参考 `fputc()`。

以下章节为参考性内容。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

另请参阅

- 2.5 标准输入输出流
- `putc()`
- XBD `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

POSIX.1-2008 技术勘误 1, XSH/TC1-2008/0471 [14] 已应用。

1.179. putchar_unlocked

SYNOPSIS (概要)

```
#include <stdio.h>

int getc_unlocked(FILE *stream);
int getchar_unlocked(void);
int putc_unlocked(int c, FILE *stream);
int putchar_unlocked(int c);
```

DESCRIPTION (描述)

应提供分别名为 `getc_unlocked()`、`getchar_unlocked()`、`putc_unlocked()` 和 `putchar_unlocked()` 的 `getc()`、`getchar()`、`putc()` 和 `putchar()` 函数版本，这些版本在功能上与原始版本等价，但例外情况是它们不需要以完全线程安全的方式实现。当在由 `flockfile()`（或 `ftrylockfile()`）和 `funlockfile()` 保护的范围内使用时，它们应该是线程安全的。当且仅当调用线程拥有 `(FILE *)` 对象时（就像成功调用 `flockfile()` 或 `ftrylockfile()` 函数后的情况），这些函数可以安全地在多线程程序中使用。

如果 `getc_unlocked()` 或 `putc_unlocked()` 作为宏实现，它们可能会多次计算 `stream` 参数，因此 `stream` 参数永远不应该是有副作用的表达式。

RETURN VALUE (返回值)

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

ERRORS (错误)

参见 `getc()`、`getchar()`、`putc()` 和 `putchar()`。

以下部分为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

由于它们可能作为宏实现，`getc_unlocked()` 和 `putc_unlocked()` 可能会错误地处理有副作用的 stream 参数。特别是，`getc_unlocked(*f++)` 和 `putc_unlocked(c,*f++)` 不一定会按预期工作。因此，在这种情况下使用这些函数前，应视情况适当执行以下语句：

```
#undef getc_unlocked
#undef putc_unlocked
```

RATIONALE (基本原理)

出于性能原因，一些I/O函数通常作为宏实现（例如，`putc()` 和 `getc()`）。为了安全，它们需要同步，但在每个字符上进行同步往往开销太大。尽管如此，安全考虑被认为更重要；因此，`getc()`、`getchar()`、`putc()` 和 `putchar()` 函数被要求是线程安全的。然而，也提供了名称明确指示其操作不安全性质的未锁定版本，但可以利用它们的高性能。这些未锁定版本只能在明确锁定的程序区域内安全使用，使用导出的锁定原语。特别是，如下序列：

```
flockfile(fileptr);
putc_unlocked('1', fileptr);
putc_unlocked('\n', fileptr);
fprintf(fileptr, "Line 2\n");
funlockfile(fileptr);
```

是允许的，并且会输出文本序列：

```
1
Line 2
```

而不会被其他线程的输出所穿插。

让像 `getc()`、`putc()` 等标准名称映射到“更快但不安全”的版本而不是“更慢但安全”的版本是错误的。无论哪种情况，在转换现有代码时，您仍然希望手动检查所有 `getc()`、`putc()` 等的使用。至少选择安全绑定作为默认值会产生正确的代码，并保持“函数原子性”不变量。否则会在转换后的代码中引入不必要的同步错误。其他修改 `stdio (FILE *)` 结构或缓冲区的例程也被安全地同步。

注意，不需要像 `getc_locked()`、`putc_locked()` 等形式的函数，因为这是 `getc()`、`putc()` 等的功能。使用特性测试宏来在 `getc_locked()` 和 `getc_unlocked()` 之间切换 `getc()` 的宏定义是不合适的，因为ISO C标准要求存在实际函数，函数的行为不能被特性测试宏改变。此外，同时提供 `xxx_locked()` 和 `xxx_unlocked()` 形式会导致混淆：后缀是描述函数的行为还是应该使用它的环境。

提供了三个额外的例程，`flockfile()`、`ftrylockfile()` 和 `funlockfile()`（它们可能是宏），允许用户勾勒出同步执行的I/O语句序列。

`ungetc()` 函数相对于其他函数/宏的调用频率较低，因此不需要未锁定变体。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- 2.5 标准I/O流
- `flockfile()`
- `getc()`
- `getchar()`
- `putc()`
- `putchar()`

XBD

CHANGE HISTORY (变更历史)

首次发布于Issue 5。为与POSIX线程扩展对齐而包含。

Issue 6

这些函数被标记为线程安全函数选项的一部分。

应用The Open Group Corrigendum U030/2，添加了描述应用程序应如何编写以避免函数作为宏实现情况的应用程序使用说明。

Issue 7

`getc_unlocked()` 、 `getchar_unlocked()` 、 `putc_unlocked()` 和 `putchar_unlocked()` 函数从线程安全函数选项移至基础规范。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0232 [395], XSH/TC1-2008/0233 [395], XSH/TC1-2008/0234 [395], 和 XSH/TC1-2008/0235 [14]。

应用POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0151 [826]。

补充信息结束。

1.180. puts

概要 (SYNOPSIS)

```
#include <stdio.h>

int puts(const char *s);
```

描述 (DESCRIPTION)

`puts()` 函数应将由 `s` 指向的字符串写入标准输出流 `stdout`，后跟一个换行符。终止的空字节不会被写入。

在 `puts()` 成功执行与在同一流上成功完成下一次 `fflush()` 或 `fclose()` 调用，或 `exit()` 或 `abort()` 调用之间，应标记文件以更新最后数据修改和最后文件状态更改时间戳。

返回值 (RETURN VALUE)

成功完成后，`puts()` 应返回一个非负数。否则，它应返回 EOF，应设置流的错误指示器，并应设置 `errno` 以指示错误。

错误 (ERRORS)

参考 `fputc()`。

以下内容为参考信息。

示例 (EXAMPLES)

打印到标准输出

以下示例获取当前时间，使用 `localtime()` 和 `asctime()` 将其转换为字符串，并使用 `puts()` 将其打印到标准输出。然后它打印正在等待的事件的分钟

数。

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
       minutes_to_event);
...
```

应用程序用法 (APPLICATION USAGE)

`puts()` 函数会附加换行符，而 `fputs()` 不会。

POSIX.1-2024 卷要求成功完成时仅返回一个非负整数。对于此要求，至少有三种已知的不同实现约定：

- 返回常数值。
- 返回最后写入的字符。
- 返回写入的字节数。请注意，对于长度超过 `{INT_MAX}` 字节的字符串，不能遵循此实现约定，因为该值无法在函数的返回类型中表示。为了向后兼容，对于长度不超过 `{INT_MAX}` 字节的字符串，实现可以返回字节数，对于所有更长的字符串，返回 `{INT_MAX}`。

原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

[2.5 标准 I/O 流, `fopen\(\)`, `fputs\(\)`, `putc\(\)`](#)

更改历史 (CHANGE HISTORY)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 7

进行了与支持细粒度时间戳相关的更改。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0476 [174,412] 和 XSH/TC1-2008/0477 [14]。

参考文本结束。

1.181. qsort, qsort_r — 数据表排序

概要

```
#include <stdlib.h>

void qsort(void *base, size_t nel, size_t width,
           int (*compar)(const void *, const void *));

[CX] void qsort_r(void *base, size_t nel, size_t width,
                  int (*compar)(const void *, const void *, void *));
```

描述

对于 `qsort()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 卷遵循 ISO C 标准。

`qsort()` 函数应对由 `nel` 个对象组成的数组进行排序，数组的首元素由 `base` 指向。每个对象的大小（以字节为单位）由 `width` 参数指定。如果 `nel` 参数值为零，则不应调用 `compar` 指向的比较函数，也不应进行任何重排。

应用程序应确保 `compar` 指向的比较函数不改变数组的内容。实现可以在比较函数调用之间重新排序数组的元素，但不应改变任何单个元素的内容。

当相同对象（由 `width` 字节组成，无论其在数组中的当前位置）被多次传递给比较函数时，结果应保持一致。也就是说，它们应定义数组上的全序关系。

数组的内容应根据比较函数按升序排序。`compar` 参数是指向比较函数的指针，该函数接收指向待比较元素的两个参数。应用程序应确保该函数返回小于、等于或大于 0 的整数，如果第一个参数被认为分别小于、等于或大于第二个参数。如果两个成员比较相等，它们在排序数组中的顺序是未指定的。

[CX] `qsort_r()` 函数应与 `qsort()` 完全相同，除了比较函数 `compar` 接受第三个参数。传递给 `qsort_r()` 的 `arg` 不透明指针应依次作为第三个参数传递给比较函数。

返回值

这些函数不应返回任何值。

错误

未定义错误。

应用程序用法

比较函数无需比较每个字节，因此元素中除了被比较的值之外，还可以包含任意数据。

如果 `compar` 回调函数需要超出待排序项目之外的任何额外状态，它只能通过全局变量访问该状态，这使得在不同线程中同时使用相同的 `compar` 函数调用 `qsort()` 可能不安全。`qsort_r()` 函数被添加，具有向比较器传递任意参数的能力，这避免了访问全局变量的需要，从而使得可以在线程间安全地共享有状态的比较器。

基本原理

要求比较函数的每个参数（下文称为 `p`）是数组元素的指针，这意味着对于每次调用，对于每个参数，以下所有表达式的值都非零：

```
((char *)p - (char *)base) % width == 0
(char *)p >= (char *)base
(char *)p < (char *)base + nel * width
```

未来方向

无。

参见

- `alphasort()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

规范性文本被更新，以避免对应用程序要求使用术语"must"。

应用 IEEE Std 1003.1-2001/Cor 1-2002，条目 XSH/TC1/D6/49，在描述部分的第一个非阴影段落添加最后一句，以及接下来的两个段落。基本原理也被更新。这些更改是为了与 ISO C 标准保持一致。

Issue 8

应用 Austin Group Defect 900，添加了 `qsort_r()` 函数。

1.182. raise

SYNOPSIS

```
#include <signal.h>

int raise(int sig);
```

DESCRIPTION

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本标准遵循 ISO C 标准。

`raise()` 函数应向执行线程或进程发送信号 `sig`。如果调用了信号处理函数，`raise()` 函数应等到信号处理函数执行完成后才返回。

`raise()` 函数的效果应等同于调用：

```
pthread_kill(pthread_self(), sig);
```

RETURN VALUE

成功完成后，应返回 0。否则，应返回非零值并设置 `errno` 来指示错误。

ERRORS

`raise()` 函数在以下情况下应失败：

- **[EINVAL]**

`sig` 参数的值是无效的信号编号。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

术语"线程"是对 ISO C 标准的扩展。

FUTURE DIRECTIONS

无。

SEE ALSO

- `kill()`
- `sigaction()`
- XBD `<signal.h>`
- XBD `<sys/types.h>`

CHANGE HISTORY

首次发布于第 4 版。衍生自 ANSI C 标准。

Issue 5

更新了 DESCRIPTION 以与 POSIX 线程扩展保持一致。

Issue 6

超出 ISO C 标准的扩展已被标记。

与单一 UNIX 规范对齐产生的 POSIX 实现新要求如下：

- 在 RETURN VALUE 部分，添加了错误时设置 `errno` 的要求。
- 添加了 [EINVAL] 错误条件。

Issue 7

与线程选项相关的功能已移至基础部分。

1.183. rand, srand — 伪随机数生成器

概要

```
#include <stdlib.h>

int rand(void);
void srand(unsigned seed);
```

描述

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`rand()` 函数应计算范围在 $[0, \{RAND_MAX\}]$ 内的伪随机整数序列，周期至少为 2^{32} 。

`rand()` 函数不需要是线程安全的；但是，`rand()` 应避免与非线程安全的伪随机序列生成函数之外的所有函数发生数据竞争。

`srand()` 函数使用参数作为新的伪随机数序列的种子，该序列将通过后续对 `rand()` 的调用返回。如果随后使用相同的种子值调用 `srand()`，则伪随机数序列将被重复。如果在任何对 `srand()` 的调用之前调用 `rand()`，则将生成与首次使用种子值 1 调用 `srand()` 时相同的序列。

`srand()` 函数不需要是线程安全的；但是，`srand()` 应避免与非线程安全的伪随机序列生成函数之外的所有函数发生数据竞争。

实现的行为应如同本 POSIX.1-2024 卷中定义的任何函数都不调用 `rand()` 或 `srand()`。

返回值

`rand()` 函数应返回序列中的下一个伪随机数。

`srand()` 函数不应返回值。

错误

未定义错误。

示例

生成伪随机数序列

以下示例演示如何生成伪随机数序列。

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* 初始化随机数生成器。 */
srand(1);

/* 仅使用小写字符创建键 */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

在不同机器上生成相同序列

以下代码定义了一对函数，可以合并到希望确保在不同机器上生成相同数字序列的应用程序中。

```
static unsigned long next = 1;

int myrand(void) /* 假设 RAND_MAX 为 32767。 */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}
```

```
void mysrand(unsigned seed)
{
    next = seed;
}
```

应用用法

当必须满足非平凡要求（包括安全性）时，应避免使用这些函数，除非使用 `getentropy()` 进行种子初始化。

`drand48()` 和 `random()` 函数提供了更精密的伪随机数生成器。

原理

ISO C 标准的 `rand()` 和 `srand()` 函数允许由所有线程共享的每进程伪随机流。这两个函数不需要改变，但必须存在互斥机制，以防止两个线程同时访问随机数生成器时的干扰。

关于 `rand()`，在多线程程序中可能需要两种不同的行为：

1. 由所有调用 `rand()` 的线程共享的单个每进程伪随机数序列
2. 每个调用 `rand()` 的线程的不同的伪随机数序列

这是通过基于种子值是整个进程全局的还是每个线程局部的来修改的线程安全函数提供的。

这没有解决 `rand()` 函数实现的已知缺陷，这些缺陷已经通过维护更多状态来解决。实际上，这指定了有缺陷函数的新线程安全形式。

未来方向

无。

参见

- [drand48\(\)](#)
 - [getentropy\(\)](#)
 - [initstate\(\)](#)
 - [<stdlib.h>](#)
-

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

包含 [rand_r\(\)](#) 函数以与 POSIX 线程扩展对齐。

添加了指出 [rand\(\)](#) 函数不需要是可重入的说明到描述部分。

Issue 6

标记了超出 ISO C 标准的扩展。

[rand_r\(\)](#) 函数被标记为线程安全函数选项的一部分。

Issue 7

应用了 Austin Group 解释 1003.1-2001 #156。

[rand_r\(\)](#) 函数被标记为过时的。

应用了 POSIX.1-2008, 技术勘误 2, XSH/TC2-2008/0301 [743]。

Issue 8

应用了 Austin Group 缺陷 1134, 添加了 [getentropy\(\)](#)。

应用了 Austin Group 缺陷 1302, 使这些函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group 缺陷 1330, 移除了过时接口。

1.184. rand, rand_r, srand - 伪随机数生成器

概要

```
#include <stdlib.h>

int rand(void);

[OB CX] int rand_r(unsigned *seed);

void srand(unsigned seed);
```

描述

对于 `rand()` 和 `srand()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2017 本卷遵循 ISO C 标准。

`rand()` 函数应计算一系列 $[0, \{RAND_MAX}\}]$ [XSI] 范围内的伪随机整数，周期至少为 2^{32} 。

[CX] `rand()` 函数不必是线程安全的。

[OB CX] `rand_r()` 函数应计算一系列 $[0, \{RAND_MAX}\}]$ 范围内的伪随机整数。 $(\{RAND_MAX\}$ 宏的值应至少为 32767。)

如果使用相同的初始值调用 `rand_r()` 指向的 `seed` 对象，并且在该对象在连续的返回和 `rand_r()` 调用之间不被修改，则应生成相同的序列。

`srand()` 函数使用参数作为新的伪随机数序列的种子，该序列将由后续的 `rand()` 调用返回。如果随后使用相同的种子值调用 `srand()`，伪随机数序列将被重复。如果在任何 `srand()` 调用之前调用 `rand()`，将生成与第一次使用种子值 1 调用 `srand()` 时相同的序列。

实现的行为应该像是 POSIX.1-2017 本卷中定义的任何函数都不会调用 `rand()` 或 `srand()`。

返回值

`rand()` 函数应返回序列中的下一个伪随机数。

[OB CX] `rand_r()` 函数应返回一个伪随机整数。

`srand()` 函数不应返回值。

错误

未定义错误。

示例

生成伪随机数序列

以下示例演示如何生成伪随机数序列。

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* 初始化随机数生成器。 */
srand(1);

/* 仅使用小写字符创建键 */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

在不同机器上生成相同序列

以下代码定义了一对函数，可以集成到希望确保在不同机器上生成相同数字序列的应用程序中。

```
static unsigned long next = 1;

int myrand(void) /* 假设 RAND_MAX 为 32767。 */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
    next = seed;
}
```

应用用法

`drand48()` 和 `random()` 函数提供了更加精密的伪随机数生成器。

在函数调用之间可以传递的状态数量的限制意味着 `rand_r()` 函数永远无法以满足伪随机数生成器所有要求的方式实现。

当需要满足非平凡要求（包括安全性）时，应避免使用这些函数。

原理

ISO C 标准的 `rand()` 和 `srand()` 函数允许所有线程共享的每进程伪随机流。这两个函数不需要改变，但必须存在互斥机制，以防止两个线程同时访问随机数生成器时的干扰。

关于 `rand()`，在多线程程序中可能需要两种不同的行为：

1. 一个单一的每进程伪随机数序列，由所有调用 `rand()` 的线程共享
2. 每个调用 `rand()` 的线程都有不同的伪随机数序列

这是通过修改的线程安全函数提供的，具体取决于种子值是对整个进程是全局的，还是对每个线程是局部的。

这并没有解决 `rand()` 函数实现中已知的缺陷，这些缺陷通过维护更多状态来处理。实际上，这指定了有缺陷函数的新线程安全形式。

未来方向

`rand_r()` 函数可能在未来的版本中被移除。

参见

`drand48()`, `initstate()`

XBD `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

为了与 POSIX 线程扩展保持一致，包含了 `rand_r()` 函数。

在 DESCRIPTION 中添加了指出 `rand()` 函数不必是可重入的注释。

Issue 6

标记了超出 ISO C 标准的扩展。

将 `rand_r()` 函数标记为线程安全函数选项的一部分。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #156。

将 `rand_r()` 函数标记为已废弃。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0301 [743]。

1.185. read

SYNOPSIS

```
#include <unistd.h>

ssize_t pread(int fildes, void *buf, size_t nbytes, off_t offset
ssize_t read(int fildes, void *buf, size_t nbytes);
```

DESCRIPTION

`read()` 函数应尝试从与打开文件描述符 `fildes` 相关联的文件中读取 `nbyte` 字节数据到 `buf` 指向的缓冲区。对同一管道、FIFO 或终端设备进行多次并发读取的行为是未指定的。

在执行下述任何操作之前，如果 `nbyte` 为零，`read()` 函数可以检测并返回如下所述的错误。在没有错误的情况下，或者如果未执行错误检测，`read()` 函数应返回零且没有其他结果。

在支持寻址的文件（例如常规文件）上，`read()` 应从与 `fildes` 相关联的文件偏移量所指定的文件位置开始读取。文件偏移量应增加实际读取的字节数。

不支持寻址的文件（例如终端）总是从当前位置读取。与此类文件相关联的文件偏移量的值是未定义的。

不会发生超过当前文件末尾的数据传输。如果起始位置在文件末尾或之后，应返回 0。如果文件引用设备特殊文件，后续 `read()` 请求的结果是实现定义的。

如果 `nbyte` 的值大于 `{SSIZE_MAX}`，结果是实现定义的。

当尝试从空管道或 FIFO 读取时：

- 如果没有进程以写方式打开管道，`read()` 应返回 0 以指示文件结束。
- 如果某个进程以写方式打开管道且设置了 `O_NONBLOCK`，`read()` 应返回 -1 并设置 `errno` 为 [EAGAIN]。
- 如果某个进程以写方式打开管道且清除了 `O_NONBLOCK`，`read()` 应阻塞调用线程，直到写入某些数据或所有以写方式打开管道的进程都关闭了管道。

当尝试读取支持非阻塞读取且当前没有可用数据的文件（管道或 FIFO 除外）时：

- 如果设置了 O_NONBLOCK，`read()` 应返回 -1 并设置 `errno` 为 [EAGAIN]。
- 如果清除了 O_NONBLOCK，`read()` 应阻塞调用线程，直到某些数据变为可用。
- 如果有可用数据，使用 O_NONBLOCK 标志没有效果。

`read()` 函数读取先前写入文件的数据。如果在常规文件的文件末尾之前的任何部分尚未被写入，`read()` 应返回值为 0 的字节。例如，`lseek()` 允许文件偏移量设置到文件中现有数据之后的位置。如果稍后在此点写入数据，则在先前数据末尾和新写入数据之间的间隙中的后续读取应返回值为 0 的字节，直到数据写入到该间隙中。

成功完成时，当 `nbyte` 大于 0，`read()` 应标记要更新文件的最后数据访问时间戳，并应返回读取的字节数。此数字永远不会大于 `nbyte`。如果文件中剩余的字节数小于 `nbyte`、`read()` 请求被信号中断、或者文件是管道或 FIFO 或特殊文件且立即可用于读取的字节数少于 `nbyte`，则返回的值可能小于 `nbyte`。例如，从与终端相关联的文件的 `read()` 可能返回一个输入行的数据。

如果在 `read()` 读取任何数据之前被信号中断，它应返回 -1 并设置 `errno` 为 [EINTR]。

如果在 `read()` 成功读取某些数据后被信号中断，它应返回已读取的字节数。

对于常规文件，不会发生超过在与 `fildes` 相关联的打开文件描述中建立的偏移量最大值的数据传输。

如果 `fildes` 引用套接字，`read()` 应等同于不设置标志的 `recv()`。

[SIO] 如果设置了 O_DSYNC 和 O_RSYNC 位，文件描述符上的读取 I/O 操作应按照同步 I/O 数据完整性完成所定义的方式完成。如果设置了 O_SYNC 和 O_RSYNC 位，文件描述符上的读取 I/O 操作应按照同步 I/O 文件完整性完成所定义的方式完成。

[SHM] 如果 `fildes` 引用共享内存对象，`read()` 函数的结果是未指定的。

[TYM] 如果 `fildes` 引用类型化内存对象，`read()` 函数的结果是未指定的。

`pread()` 函数应等同于 `read()`，不同之处在于它应从文件中的给定位置读取而不改变文件偏移量。`pread()` 的前三个参数与 `read()` 相同，增加了第四个参数 `offset` 用于指定文件内期望的位置。尝试对无法寻址的文件执行 `pread()` 将导致错误。

RETURN VALUE

成功完成时，这些函数应返回一个非负整数，指示实际读取的字节数。否则，函数应返回 -1 并设置 `errno` 以指示错误。

ERRORS

这些函数在以下情况下可能失败：

[EAGAIN]

文件既不是管道，也不是 FIFO，也不是套接字，文件描述符设置了 `O_NONBLOCK` 标志，且线程将在读取操作中延迟。

[EBADF]

`fd` 参数不是一个有效的用于读取的打开文件描述符。

[EINTR]

读取操作由于收到信号而终止，且没有数据被传输。

[EIO]

进程是后台进程组的成员，试图从其控制终端读取，并且调用线程正在阻塞 `SIGTTIN`，或者进程正在忽略 `SIGTTIN`，或者进程的进程组是孤儿进程组。此错误也可能由于实现定义的原因而生成。

[EISDIR]

[XSI] `fd` 参数引用目录，且实现不允许使用 `read()` 或 `pread()` 读取目录。应使用 `readdir()` 函数代替。

[EOVERFLOW]

文件是常规文件，`nbyte` 大于 0，起始位置在文件结束之前，且起始位置大于或等于与 `fd` 相关联的打开文件描述符中建立的偏移量最大值。

`pread()` 函数在以下情况下可能失败：

[EINVAL]

文件是常规文件或块特殊文件，且 `offset` 参数为负数。文件偏移量应保持不变。

[ESPIPE]

文件无法寻址。

`read()` 函数在以下情况下可能失败：

[EAGAIN]

文件是管道或 FIFO，文件描述符设置了 `O_NONBLOCK` 标志，且线程将在读取操作中延迟。

[EAGAIN] 或 [EWOULDBLOCK]

文件是套接字，文件描述符设置了 O_NONBLOCK 标志，且线程将在读取操作中延迟。

[ECONNRESET]

尝试在套接字上读取，但连接被其对等方强制关闭。

[ENOTCONN]

尝试在未连接的套接字上读取。

[ETIMEDOUT]

尝试在套接字上读取，但发生了传输超时。

这些函数在以下情况下可能失败：

[EIO]

发生了物理 I/O 错误。

[ENOBUFS]

系统中可用资源不足，无法执行操作。

[ENOMEM]

可用内存不足，无法满足请求。

[ENXIO]

对不存在的设备发出了请求，或者请求超出了设备的能力。

EXAMPLES

将数据读入缓冲区

以下示例将数据从与文件描述符 `fd` 相关联的文件读取到 `buf` 指向的缓冲区中。

```
#include <sys/types.h>
#include <unistd.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_read;
int fd;
...
nbytes = sizeof(buf);
bytes_read = read(fd, buf, nbytes);
...
```

APPLICATION USAGE

无。

RATIONALE

POSIX.1-2024 的这一卷没有指定返回错误后的文件偏移量值；情况太多。对于编程错误，如 [EBADF]，由于不涉及文件，这个概念没有意义。对于立即检测到的错误，如 [EAGAIN]，显然偏移量不应该改变。然而，在中断或硬件错误之后，更新后的值将非常有用，这也是许多实现的行为。

请注意，读取零字节的 `read()` 不会修改最后数据访问时间戳。请求读取超过零字节但返回零的 `read()` 需要修改最后数据访问时间戳。

允许实现执行对零字节 `read()` 请求的错误检查，但不是必需的。

输入和输出

使用大字节计数的 I/O 总是带来问题。诸如 `lread()` 和 `lwrite()`（使用和返回 `long`）的想法曾被考虑过。当前的解决方案是在 ISO C 标准函数上使用抽象类型来处理 `read()` 和 `write()`。可以声明抽象类型，以便现有函数工作，也可以声明抽象类型，以便在未来的实现中表示更大的类型。据推测，限制 `size_t` 最大范围的任何约束也将限制可移植 I/O 请求到相同范围。

POSIX.1-2024 的这一卷还通过要求字节计数被限制以使有符号返回值保持有意义来进一步限制范围。由于返回类型也是（有符号的）抽象类型，字节计数可以由实现定义为大于 `int` 可以容纳的值。

标准制定者考虑为管道或 FIFO 添加原子性要求，但认识到由于管道和 FIFO 的性质，无法保证对 {PIPE_BUF} 或任何其他大小的读取的原子性，而这将有助于应用程序的可移植性。

POSIX.1-2024 的这一卷要求当 `nbyte` 为零时，`read()` 或 `write()` 不采取任何操作。这并不意图优先于错误检测（如无效的缓冲区指针或文件描述符）。这与 POSIX.1-2024 的这一卷的其余部分一致，但这里的措辞可能被误读为要求在检测到任何其他错误之前检测零情况。零值应被视为正确的值，其语义是无操作。

I/O 意图对常规文件、管道和 FIFO 是原子的。原子意味着单个操作中一起开始的所有字节最终一起结束，没有其他 I/O 操作的交错。这是终端的一个已知属性不被遵守，终端被明确地（和隐含地永久地）排除，使得行为未指定。其他设备类型的行为也未指定，但措辞意图暗示未来的标准可能选择指定原子性（或不指定）。

有建议向 `read()` 和 `write()` 添加格式参数，以处理异构文件系统和基本硬件类型之间的网络传输。这样的设施可能是 OSI 表示层服务支持所需要的。然而，确定这应该与类似的 C 语言设施相对应，而这超出了 POSIX.1-2024 的这一卷的范围。这个概念被建议给 ISO C 标准的开发者考虑，作为未来工作的可能领域。

在 4.3 BSD 中，在传输任何数据之前被信号中断的 `read()` 或 `write()` 默认不返回 [EINTR] 错误，而是被重新启动。在 4.2 BSD、4.3 BSD 和第八版中，有一个额外的函数 `select()`，其目的是暂停直到在指定的文件描述符上检测到指定的活动（要读取的数据、要写入的空间等）。在为这些系统编写的应用程序中，在由于信号而中断 I/O 的情况（如键盘输入）下，在 `read()` 之前使用 `select()` 是常见的。

哪些文件或文件类型可中断的问题被认为是实现设计问题。这通常主要受硬件和可靠性问题的影响。

没有对"不可恢复错误"之后采取的行动的引用。这被认为超出了 POSIX.1-2024 的这一卷描述硬件错误情况时发生什么的范围。

本标准的早期版本允许两种非常不同的中断处理行为。为了尽量减少由此产生的混淆，决定 POSIX.1-2024 应只支持其中一种行为。在 AT&T 衍生系统上的历史实践是，当在部分但不是所有请求的数据被传输后被中断时，`read()` 和 `write()` 返回 -1 并设置 `errno` 为 [EINTR]。然而，美国商务部 FIPS 151-1 和 FIPS 151-2 要求历史 BSD 行为，其中 `read()` 和 `write()` 返回中断前实际传输的字节数。如果在传输任何数据时返回 -1，在可寻址设备上很难从错误中恢复，在不可寻址设备上是不可能的。大多数新实现都支持这种行为。POSIX.1-2024 要求的行为是返回传输的字节数。

POSIX.1-2024 没有指定缓冲 `read()` 的实现何时实际将数据移动到用户提供的缓冲区，因此实现可以选择在最晚的时刻执行此操作。因此，较早到达的中断可能不会导致 `read()` 返回部分字节计数，而是返回 -1 并设置 `errno` 为 [EINTR]。

还考虑了结合前两个选项，设置 `errno` 为 [EINTR] 同时返回短计数。然而，不仅没有现有的实践实现这一点，它也与当设置 `errno` 时，负责的函数应返回 -1 的想法相矛盾。

POSIX.1-2024 的这一卷故意没有指定任何与管道、FIFO 和套接字相关的 `pread()` 错误，除了 [ESPIPE]。

FUTURE DIRECTIONS

无。

SEE ALSO

`fcntl()`, `lseek()`, `open()`, `pipe()`, `readv()`

XBD 11. General Terminal Interface, `<sys/uio.h>`, `<unistd.h>`

CHANGE HISTORY

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

DESCRIPTION 更新以与 POSIX Realtime Extension 和 POSIX Threads Extension 对齐。

添加了 Large File Summit 扩展。

添加了 `pread()` 函数。

Issue 6

DESCRIPTION 和 ERRORS 部分更新，使得对 STREAMS 的引用被标记为 XSI STREAMS Option Group 的一部分。

以下对 POSIX 实现的新要求源自与 Single UNIX Specification 的对齐：

- DESCRIPTION 现在说明如果 `read()` 在成功读取某些数据后被信号中断，它返回读取的字节数。在 Issue 3 中，`read()` 是返回读取的字节数还是返回 -1 并设置 `errno` 为 [EINTR] 是可选的。这是一个 FIPS 要求。
- 在 DESCRIPTION 中，添加了文本以指示对于常规文件，不会发生超过在与 `fildes` 相关联的打开文件描述中建立的偏移量最大值的数据传输。此更改是为了支持大文件。
- 添加了 [EOVERFLOW] 强制错误条件。
- 添加了 [ENXIO] 可选错误条件。

添加了引用套接字的文本到 DESCRIPTION 中。

进行了以下更改以与 IEEE P1003.1a 草案标准对齐：

- 澄清了读取零字节的效果。

DESCRIPTION 通过指定对类型化内存对象的 `read()` 结果是未指定的来更新以与 IEEE Std 1003.1j-2000 对齐。

添加了新的 RATIONALE 来解释输入和输出操作的原子性要求。

为套接字操作添加了以下错误条件：[EAGAIN]、[ECONNRESET]、[ENOTCONN] 和 [ETIMEDOUT]。

[EIO] 错误设为可选。

为套接字操作添加了以下错误条件：[ENOBUFS] 和 [ENOMEM]。

`readv()` 函数被拆分为单独的参考页面。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/108，更新 ERRORS 部分中的 [EAGAIN] 错误，从“进程将被延迟”改为“线程将被延迟”。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/109，在 RATIONALE 部分进行了编辑修正。

Issue 7

`pread()` 函数从 XSI 选项移动到 Base。

与 XSI STREAMS 选项相关的功能被标记为过时。

进行了与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0480 [218], XSH/TC1-2008/0481 [79], XSH/TC1-2008/0482 [218], XSH/TC1-2008/0483 [218], XSH/TC1-2008/0484 [218], 和 XSH/TC1-2008/0485 [218,428]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0302 [710] 和 XSH/TC2-2008/0303 [676,710]。

Issue 8

应用 Austin Group Defect 1330，移除过时接口。

1.186. realloc, reallocarray — 内存重新分配器

概要

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size);

[CX] void *reallocarray(void *ptr, size_t nelem, size_t elsize)
```

描述

对于 `realloc()`：[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非预期的。POSIX.1-2024 本卷遵循 ISO C 标准。

`realloc()` 函数应该释放 `ptr` 指向的旧对象，并返回一个指向具有 `size` 指定大小的新对象的指针。新对象的内容在释放前应与旧对象的内容相同，直到新旧大小中的较小者为止。新对象中超出旧对象大小的任何字节都具有不确定的值。

[CX] `reallocarray()` 函数应等同于调用 `realloc(ptr, nelem * elsize)`，不同之处在于乘法溢出应为错误。

如果 `ptr` 是空指针，`realloc()` [CX] 或 `reallocarray()` 应等同于为指定大小调用 `malloc()`。否则，如果 `ptr` 与之前由 `aligned_alloc()`、`calloc()`、`malloc()`、[ADV] `posix_memalign()`、`realloc()`、[CX] `reallocarray()` 或 POSIX.1-2024 中如同通过 `malloc()` 分配内存的函数返回的指针不匹配，或者如果空间已经通过调用 `free()`、[CX] `reallocarray()` 或 `realloc()` 被释放，则行为未定义。

如果 `size` 非零且新对象的内存未分配，则旧对象不应被释放。

通过连续调用 `realloc()` [CX] 或 `reallocarray()` 分配的存储的顺序和连续性是未指定的。如果分配成功，返回的指针应适当对齐，以便可以将其赋值给具有基本对齐要求的任何类型对象的指针，然后用于访问分配空间中的此类对象（直到空间被显式释放或重新分配）。每次此类分配应产生一个与任何其他对象不相交的对象的指针。返回的指针应指向分配空间的起始位置（最低字节地址）。如果无法分配空间，应返回空指针。

为了确定数据竞争的存在，`realloc()` [CX] 和 `reallocarray()` 的行为应如同只访问通过其参数可访问的内存位置，而不访问其他静态持续存储。然而，函数可以明显修改其分配的存储。对特定内存区域进行分配或释放的 `aligned_alloc()` 、`calloc()` 、`free()` 、`malloc()` 、 [ADV] `posix_memalign()` 、 [CX] `reallocarray()` 和 `realloc()` 调用应发生在单一总顺序中（参见 4.15.1 内存排序），并且每个此类释放调用应与此顺序中的下一个分配（如果有）同步。

返回值

成功完成时，`realloc()` [CX] 和 `reallocarray()` 应返回指向新对象的指针（该值可以与指向旧对象的指针相同），如果新对象未分配，则返回空指针。

[OB] 如果 `size` 为 0，[OB CX] 或 `nelem` 或 `elsize` 为 0，[OB] 则：

如果没有足够的可用内存，`realloc()` [CX] 和 `reallocarray()` 应返回空指针 [CX] 并将 `errno` 设置为 [ENOMEM]。

错误

如果出现以下情况，`realloc()` [CX] 和 `reallocarray()` 函数应失败：

[ENOMEM]

[CX] 可用内存不足。

[CX] 如果出现以下情况，`reallocarray()` 函数应失败：

[ENOMEM]

计算 `nelem * elsize` 会溢出。

如果出现以下情况，`realloc()` [CX] 和 `reallocarray()` 函数可能失败：

[EINVAL]

[CX] 请求的分配大小为 0 且实现不支持大小为 0 的分配。

示例

无。

应用程序用法

ISO C 标准规定了 `realloc(p, 0)` 调用在因新对象的内存未分配而返回空指针时是否释放 `p` 指向的空间是由实现定义的。POSIX.1 反而要求如果返回空指针且空间未被释放，实现应设置 `errno`，而 POSIX 应用程序应仅在 `errno` 被更改时才释放空间。

基本原理

参见 `malloc()` 的基本原理。

未来方向

ISO C 标准指出，使用等于零的 `size` 参数调用 `realloc()` 是一个已过时的特性。此特性可能在标准的未来版本中被移除。

另请参见

- `aligned_alloc()`
- `calloc()`
- `free()`
- `malloc()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 部分，如果没有足够的可用内存，添加了将 `errno` 设置为 [ENOMEM]。

- 添加了 [ENOMEM] 错误条件。

Issue 7

应用了 POSIX.1-2008、技术勘误 1、XSH/TC1-2008/0495 [400]、XSH/TC1-2008/0496 [400]、XSH/TC1-2008/0497 [400] 和 XSH/TC1-2008/0498 [400]。

应用了 POSIX.1-2008、技术勘误 2、XSH/TC2-2008/0309 [526] 和 XSH/TC2-2008/0310 [526,688]。

Issue 8

应用了 Austin Group 缺陷 374，添加了 [EINVAL] 错误。

应用了 Austin Group 缺陷 1218，添加了 `reallocarray()`。

应用了 Austin Group 缺陷 1302，使 `realloc()` 函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group 缺陷 1387，更改了基本原理部分。

1.187. scanf

概要

```
#include <stdio.h>

int scanf(const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int sscanf(const char *s, const char *format, ...);
```

描述

`fscanf()` 函数应从指定的输入流 `stream` 读取。`scanf()` 函数应从标准输入流 `stdin` 读取。`sscanf()` 函数应从字符串 `s` 读取。每个函数读取字节，根据格式解释它们，并将结果存储在其参数中。每个函数期望作为参数一个控制字符串 `format` (如下所述)，以及一组指向应该存储转换输入位置的 `指针` 参数。如果格式没有足够的参数，则结果未定义。如果格式耗尽而参数仍有剩余，则应评估多余的参数，但否则忽略。

转换可以应用于参数列表中格式之后的第 n 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 % (见下文) 被序列 "%n\$" 替换，其中 n 是范围 [1,{NL_ARGMAX}] 内的十进制整数。此功能提供了定义以适合特定语言的顺序选择参数的格式字符串。在包含 "%n\$" 形式转换说明的格式字符串中，未指定是否可以从格式字符串多次引用参数列表中的编号参数。

格式可以包含任一形式的转换说明——即 % 或 "%n\$"——但两种形式不能在单个格式字符串中混合使用。唯一的例外是 %% 或 %* 可以与 "%n\$" 形式混合使用。使用编号参数规范时，指定第 N 个参数要求所有前导参数，从第一个到第 $(N-1)$ 个，都是指针。

所有形式的 `fscanf()` 函数都应允许在输入字符串中检测依赖于语言的小数字符。小数字符在当前语言环境 (类别 LC_NUMERIC) 中定义。在 POSIX 语言环境中，或在未定义小数字符的语言环境中，小数字符应默认为句点 ('.')。

格式字符串

应用程序应确保格式是一个字符字符串，如果有初始转换状态，则以其开始和结束，由零个或多个指令组成。每个指令由以下内容之一组成：一个或多个空白字节；一个普通字符 (既不是 '%' 也不是空白字节)；或一个转换说明。每个转换说明由字符 '%' 或字符序列 "%n\$" 引入，之后依次出现以下内容：

- 可选的赋值抑制字符 '*'。
- 可选的非零十进制整数，指定最大字段宽度。
- 可选的赋值分配字符 'm'。
- 可选的长度修饰符，指定接收对象的大小。
- 转换说明符字符，指定要应用的转换类型。有效的转换说明符如下所述。

执行过程

`fscanf()` 函数应依次执行格式的每个指令。当所有指令都已执行，或者如果指令失败（如下详述），函数应返回。失败被描述为输入失败（由于输入字节不可用）或匹配失败（由于输入不合适）。

由一个或多个空白字节组成的指令应通过读取输入直到第一个非空白字节来执行，该字节应保持未读状态，或者直到无法读取更多字节。该指令永不失败。

作为普通字符的指令应按以下方式执行：应从输入中读取下一个字节并与构成指令的字节进行比较；如果比较表明它们不等效，则指令应失败，差异和后续字节应保持未读状态。类似地，如果文件结束、编码错误或读取错误阻止读取字符，则指令应失败。

作为转换说明的指令定义了一组匹配输入序列，如下所述为每个转换字符描述。转换说明应按以下步骤执行。

应跳过输入空白字节，除非转换说明包含 [、c、C 或 n 转换说明符。

应从输入中读取项，除非转换说明包含 n 转换说明符。输入项应定义为最长输入字节序列（最多到任何指定的最大字段宽度，可能以字符或字节为单位测量，取决于转换说明符），它是匹配序列的初始子序列。输入项之后的第一个字节（如果有）应保持未读状态。如果输入项的长度为 0，则转换说明的执行应失败；此条件是匹配失败，除非文件结束、编码错误或读取错误阻止从流输入，在这种情况下是输入失败。

除了 % 转换说明符的情况下，输入项（或者在 %n 转换说明的情况下，输入字节的数量）应转换为适合转换字符的类型。如果输入项不是匹配序列，则转换说明的执行失败；此条件是匹配失败。除非由 '*' 指示赋值抑制，转换的结果应放置在格式参数之后第一个尚未接收转换结果的参数指向的对象中（如果转换说明由 % 引入），或放在第 n 个参数中（如果由字符序列 "%n\$" 引入）。如果此对象没有适当的类型，或者转换结果无法在提供的空间中表示，则行为未定义。

赋值分配

c、s 和 [转换说明符应接受可选的赋值分配字符 'm'，这将导致分配内存缓冲区来保存转换结果。如果转换说明符是 s 或 [，分配的缓冲区应包括终止空字符（或宽字符）的空间。在这种情况下，对应于转换说明符的参数应该是指向指针变量的引用，该变量将接收指向分配缓冲区的指针。系统应分配缓冲区，如同调用了 `malloc()`。应用程序应负责在使用后释放内存。如果没有足够的内存来分配缓冲区，函数应将 `errno` 设置为 [ENOMEM] 并导致转换错误。如果函数返回 EOF，则此调用使用赋值分配字符 'm' 为参数成功分配的任何内存在函数返回前应被释放。

长度修饰符

长度修饰符及其含义是：

hh

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `signed char` 或 `unsigned char` 的参数。

h

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `short` 或 `unsigned short` 的参数。

l (ell)

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `long` 或 `unsigned long` 的参数；后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于指向 `double` 的参数；或后续的 c、s 或 [转换说明符应用于指向 `wchar_t` 的参数。如果指定了 'm' 赋值分配字符，则转换应用于指向指向 `wchar_t` 的指针的参数。

ll (ell-ell)

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `long long` 或 `unsigned long long` 的参数。

j

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `intmax_t` 或 `uintmax_t` 的参数。

z

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `size_t` 或相应有符号整数类型的参数。

t

指定后续的 d、i、o、u、x、X 或 n 转换说明符应用于指向 `ptrdiff_t` 或相应 `unsigned` 类型的参数。

L

指定后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于指向 **long double** 的参数。

如果长度修饰符出现在除上述指定之外的任何转换说明符中，则行为未定义。

转换说明符

以下转换说明符是有效的：

d

匹配可选有符号十进制整数，其格式与预期 `strtol()` 的基数为 10 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **int** 的指针。

i

匹配可选有符号整数，其格式与预期 `strtol()` 的基数为 0 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **int** 的指针。

o

匹配可选有符号八进制整数，其格式与预期 `strtoul()` 的基数为 8 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **unsigned** 的指针。

u

匹配可选有符号十进制整数，其格式与预期 `strtoul()` 的基数为 10 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **unsigned** 的指针。

x

匹配可选有符号十六进制整数，其格式与预期 `strtoul()` 的基数为 16 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **unsigned** 的指针。

a, e, f, g

匹配可选有符号浮点数、无穷大或 NaN，其格式与预期 `strtod()` 的主题序列相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 **float** 的指针。

如果 `fprintf()` 函数族为无穷大和 NaN（以浮点格式编码的符号实体）生成字符字符串表示以支持 IEEE Std 754-1985，则 `fscanf()` 函数族应将它们识别为输入。

s

匹配非空白字节序列。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数

是指向 **char**、**signed char** 或 **unsigned char** 数组的初始字节的指针，该数组足够大以接受序列和自动添加的终止空字符代码。否则，应用程序应确保相应参数是指向指向 **char** 的指针。

如果存在 l (ell) 限定符，输入是开始于初始转换状态的字符序列。每个字符应转换为宽字符，如同通过调用 **mbrtowc()** 函数，转换状态由在第一个字符转换前初始化为零的 **mbstate_t** 对象描述。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数是指向 **wchar_t** 数组的指针，该数组足够大以接受序列和自动添加的终止空宽字符。否则，应用程序应确保相应参数是指向指向 **wchar_t** 的指针。

[

匹配来自预期字节集合（扫描集）的非空字节序列。在这种情况下应抑制正常跳过空白字节。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数是指向 **char**、**signed char** 或 **unsigned char** 数组的初始字节的指针，该数组足够大以接受序列和自动添加的终止空字节。否则，应用程序应确保相应参数是指向指向 **char** 的指针。

如果存在 l (ell) 限定符，输入是开始于初始转换状态的字符序列。序列中的每个字符应转换为宽字符，如同通过调用 **mbrtowc()** 函数，转换状态由在第一个字符转换前初始化为零的 **mbstate_t** 对象描述。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数是指向 **wchar_t** 数组的指针，该数组足够大以接受序列和自动添加的终止空宽字符。否则，应用程序应确保相应参数是指向指向 **wchar_t** 的指针。

转换说明包括格式字符串中直到并包括匹配右方括号 (']') 的所有后续字节。方括号之间的字节（扫描列表）构成扫描集，除非左方括号之后的字节是插入符号 (^)，在这种情况下扫描集包含所有不出现在插入符号和右方括号之间扫描列表中的字节。如果转换说明以 "[]" 或 "[^]" 开始，右方括号包含在扫描列表中，下一个右方括号是结束转换说明的匹配右方括号；否则，第一个右方括号是结束转换说明的那个。如果 '^' 在扫描列表中且不是第一个字符，也不是第一个字符为 '^' 时的第二个字符，也不是最后一个字符，则行为是实现定义的。

c

匹配由字段宽度指定的数字数量的字节序列（如果转换说明中没有字段宽度则为 1）。不添加空字节。在这种情况下应抑制正常跳过空白字节。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数是指向 **char**、**signed char** 或 **unsigned char** 数组的初始字节的指针，该数组足够大以接受序列。否则，应用程序应确保相应参数是指向指向 **char** 的指针。

如果存在 l (ell) 限定符，输入应是开始于初始转换状态的字符序列。序列中的每个字符转换为宽字符，如同通过调用 **mbrtowc()** 函数，转换状态由在第一个字符转换前初始化为零的 **mbstate_t** 对象描述。不添加空宽字符。如果未指定 'm' 赋值分配字符，应用程序应确保相应参数是指向 **wchar_t** 数组的指针，该

数组足够大以接受生成的宽字符序列。否则，应用程序应确保相应参数是指向指向 **wchar_t** 的指针。

p

匹配实现定义的序列集合，该集合应与相应 **fprintf()** 函数的 %p 转换说明产生的序列集合相同。应用程序应确保相应参数是指向指向 **void** 的指针。输入项的解释是实现定义的。如果输入项是在同一程序执行期间先前转换的值，则结果指针应与该值比较相等；否则，%p 转换说明的行为未定义。

n

不消耗输入。应用程序应确保相应参数是指向整数的指针，其中应写入到目前为止通过此调用 **fscanf()** 函数从输入读取的字节数。%n 转换说明的执行不应增加函数执行完成时返回的赋值计数。不转换参数，但应消耗一个。如果转换说明包含赋值抑制字符或字段宽度，则行为未定义。

c

等效于 lc。

s

等效于 ls。

%

匹配单个 '%' 字符；不发生转换或赋值。完整的转换说明应为 %%。

如果转换说明无效，则行为未定义。

转换说明符 A、E、F、G 和 X 也有效，并应分别等效于 a、e、f、g 和 x。

终止条件

如果在输入期间遇到文件结束，则应终止转换。如果在读取任何匹配当前转换说明（除了 %n）的字节（除了允许的前导空白字节）之前发生文件结束，则当前转换说明的执行应以输入失败终止。否则，除非当前转换说明的执行以匹配失败终止，则后续转换说明（如果有）的执行应以输入失败终止。

在 **sscanf()** 中到达字符串末尾应等效于 **fscanf()** 遇到文件结束。

如果转换在冲突输入上终止，则违规输入保留在输入中未读。任何尾随空白字节（包括换行字符）应保持未读状态，除非由转换说明匹配。文字匹配和抑制赋值的成功只能通过 %n 转换说明直接确定。

fscanf() 和 **scanf()** 函数可能标记与 **stream** 关联的文件的最后数据访问时间戳以进行更新。第一次成功执行使用 **stream** 并返回非先前 **ungetc()** 调用提供的数据的 **fgetc()**、**fgets()**、**fread()**、**getc()**、**getchar()**、**getdelim()**、**getline()**、**fscanf()** 或 **scanf()** 时，最后数据访问时间戳应标记为更新。

返回值

成功完成后，这些函数应返回成功匹配和赋值的输入项数量；在早期匹配失败的情况下此数量可以为零。如果输入在第一次匹配失败或转换之前结束，应返回 EOF。如果发生读取错误，应设置流的错误指示符，应返回 EOF，并应设置 `errno` 以指示错误。

错误

函数在以下情况下失败：

- **EAGAIN** - 为 `stream` 底层的文件描述符设置了 O_NONBLOCK 标志，并且线程将在 `fscanf()` 操作中被延迟。
- **EBADF** - `stream` 底层的文件描述符不是为读取而打开的有效文件描述符。
- **EINTR** - 读操作由于接收信号而终止，且未传输数据。
- **EIO** - 发生物理 I/O 错误。
- **ENOMEM** - 存储空间不足。

`fscanf()` 和 `scanf()` 函数在以下情况下可能失败：

- **OVERFLOW** - 文件是常规文件，并且尝试在相应流的偏移量最大值处或超出该值进行读取。

`sscanf()` 函数在以下情况下应失败：

- **ENAMETOOLONG** - 路径名组件的长度超过 {NAME_MAX}。

示例

```
#include <stdio.h>

int main() {
    int i, j;
    float x, y;
    char s[80];

    /* 基本用法 */
    scanf("%d %f", &i, &x);

    /* 读入多个变量 */
}
```

```
scanf("%d %f %s", &j, &y, s);

/* 使用赋值抑制 */
scanf("%*d %d", &i); /* 跳过第一个整数, 读取第二个 */

/* 使用字段宽度 */
scanf("%4d %4d", &i, &j); /* 读取两个 4 位整数 */

return 0;
}
```

应用程序用法

鼓励程序员使用 `fgets()` 和 `sscanf()` 而不是 `scanf()`，以避免缓冲区溢出问题并更优雅地处理输入错误。

基本原理

`scanf()` 函数族提供格式化输入功能，补充 `printf()` 格式化输出函数族。赋值分配功能 ('m' 字符) 被添加以简化可变长度字符串输入的处理。

未来方向

无。

另见

- `fgets()`
- `fprintf()`
- `getc()`
- `getdelim()`
- `getline()`
- `malloc()`
- `setlocale()`
- `strtod()`

- `strtol()`
- `strtoul()`

版权

本文的部分内容从 IEEE Std 1003.1-2024, 信息技术标准 -- 便携式操作系统接口 (POSIX), The Open Group 基本规范第 7 版, 版权所有 (C) 2024 由电气和电子工程师协会有限公司和 The Open Group。如果此版本与原始 IEEE 和 The Open Group 标准之间存在任何差异, 原始 IEEE 和 The Open Group 标准为裁判 文 档 。原 始 标 准 可 在 线 获 取 于 <http://www.opengroup.org/unix/online.html>。

本页中出现的任何印刷或格式错误很可能是转录原始文档信息时的错误, 不是标准本身的一部分。

1.188. sched_get_priority_max, sched_get_priority_min

SYNOPSIS (概要)

```
[PS|TPS] #include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION (描述)

`sched_get_priority_max()` 和 `sched_get_priority_min()` 函数应分别返回由 `policy` 指定的调度策略的相应最大值或最小值。

`policy` 的值应为 `<sched.h>` 中定义的调度策略值之一。

RETURN VALUE (返回值)

如果成功, `sched_get_priority_max()` 和 `sched_get_priority_min()` 函数应分别返回相应的最大值或最小值。如果不成功, 它们应返回值 -1 并设置 `errno` 以指示错误。

ERRORS (错误)

如果出现以下情况, `sched_get_priority_max()` 和 `sched_get_priority_min()` 函数应失败:

- `EINVAL`
- `policy` 参数的值不表示已定义的调度策略。

以下部分为参考信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `sched_getparam()`
- `sched_setparam()`
- `sched_getscheduler()`
- `sched_rr_get_interval()`
- `sched_setscheduler()`

XBD `<sched.h>`

CHANGE HISTORY (变更历史)

首次在 Issue 5 中发布。为与 POSIX 实时扩展保持一致而包含。

Issue 6

这些函数被标记为进程调度选项的一部分。

如果实现不支持进程调度选项，则不需要提供存根，因此已删除 [ENOSYS] 错误条件。

由于这些函数不接受 `pid` 参数，因此已删除 [ESRCH] 错误条件。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/52，将 SYNOPSIS 中的 PS 边际代码更改为 PS|TPS。

1.189. `sched_get_priority_max`, `sched_get_priority_min`

SYNOPSIS (概要)

```
#include <sched.h>

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

DESCRIPTION (描述)

`sched_get_priority_max()` 和 `sched_get_priority_min()` 函数应分别返回由 `policy` 指定的调度策略所对应的最大值或最小值。

`policy` 的值应为 `<sched.h>` 中定义的调度策略值之一。

RETURN VALUE (返回值)

如果成功, `sched_get_priority_max()` 和 `sched_get_priority_min()` 函数应分别返回相应的最大值或最小值。如果不成功, 它们应返回值 -1 并设置 `errno` 来指示错误。

ERRORS (错误)

`sched_get_priority_max()` 和 `sched_get_priority_min()` 函数在以下情况下应失败:

- `EINVAL`
- `policy` 参数的值不表示已定义的调度策略。

以下部分为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `sched_getparam()`
- `sched_setparam()`
- `sched_getscheduler()`
- `sched_rr_get_interval()`
- `sched_setscheduler()`

XBD `<sched.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

这些函数被标记为进程调度选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持进程调度选项，不需要提供存根。

[ESRCH] 错误条件已被移除，因为这些函数不接受 **pid** 参数。

应用 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/52，将 SYNOPSIS 中的 PS 边距代码更改为 PS|TPS。

1.190. sched_rr_get_interval — 获取执行时间限制 (REALTIME)

概要

```
#include <sched.h>

int sched_rr_get_interval(pid_t pid, struct timespec *interval)
```

描述

`sched_rr_get_interval()` 函数应更新由 `interval` 参数引用的 `timespec` 结构体，使其包含由 `pid` 指定的进程的当前执行时间限制（即时量程）。如果 `pid` 为零，则应返回调用进程的当前执行时间限制。

返回值

如果成功，`sched_rr_get_interval()` 函数应返回零。否则，它应返回值 -1 并设置 `errno` 来指示错误。

错误

`sched_rr_get_interval()` 函数在以下情况下应失败：

- **[ESRCH]** - 找不到与 `pid` 指定的进程相对应的进程。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参见

- `sched_getparam()`
- `sched_get_priority_max()`
- `sched_getscheduler()`
- `sched_setparam()`
- `sched_setscheduler()`

XBD `<sched.h>`

变更历史

在 Issue 5 中首次发布。为与 POSIX 实时扩展保持一致而包含在内。

Issue 6

`sched_rr_get_interval()` 函数被标记为进程调度选项的一部分。

`[ENOSYS]` 错误条件已被移除，因为如果实现不支持进程调度选项，则无需提供存根。

应用 IEEE Std 1003.1-2001/Cor 1-2002, XSH/TC1/D6/53，将概要中的 PS 边际代码更改为 PS|TPS。

1.191. sem_close — 关闭命名信号量

SYNOPSIS

```
#include <semaphore.h>

int sem_close(sem_t *sem);
```

DESCRIPTION

`sem_close()` 函数应表示调用进程已完成使用由 `sem` 指示的命名信号量。对无名信号量（由 `sem_init()` 创建的信号量）调用 `sem_close()` 的效果是未定义的。`sem_close()` 函数应释放（即使其可用于此进程后续的 `sem_open()` 调用重用）系统为此进程使用此信号量分配的任何系统资源。如果由 `sem` 指示的信号量是使用文件描述符实现的，则文件描述符应被关闭。此进程后续使用由 `sem` 指示的信号量的效果是未定义的。如果调用进程中的任何线程当前在信号量上被阻塞，则行为是未定义的。如果信号量尚未通过成功的 `sem_unlink()` 调用移除，那么 `sem_close()` 对信号量的状态没有影响。如果在此信号量最近一次使用 `O_CREAT` 调用 `sem_open()` 后，已成功为 `name` 调用了 `sem_unlink()` 函数，那么当所有打开该信号量的进程关闭所有信号量句柄时，该信号量将不再可访问。

RETURN VALUE

成功完成时，应返回零值。否则，应返回 -1 值并设置 `errno` 来指示错误。

ERRORS

`sem_close()` 函数可能在以下情况下失败：

- **[EINVAL]**

`sem` 参数不是有效的信号量描述符。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

`semctl()` , `semget()` , `semop()` , `sem_init()` , `sem_open()` ,
`sem_unlink()`

XBD `<semaphore.h>`

CHANGE HISTORY

Issue 6

`sem_close()` 函数被标记为信号量选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持信号量选项，则不需要提供存根。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/113，更新 ERRORS 部分，使 [EINVAL] 错误变为可选的。

Issue 7

`sem_close()` 函数从信号量选项移动到基础。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0523 [37]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0317 [870]。

Issue 8

应用 Austin Group Defect 368, 添加了如果 `sem` 使用文件描述符实现, 则 `sem_close()` 关闭文件描述符的要求。

应用 Austin Group Defect 1324, 阐明了已取消链接的信号量不再可访问的情况。

1.192. sem_destroy — 销毁无名信号量

概要

```
#include <semaphore.h>

int sem_destroy(sem_t *sem);
```

描述

`sem_destroy()` 函数应当销毁由 `sem` 指示的无名信号量。如果无名信号量使用文件描述符实现，则文件描述符应当被关闭。只有使用 `sem_init()` 创建的信号量才能使用 `sem_destroy()` 销毁；使用有名信号量调用 `sem_destroy()` 的效果是未定义的。在 `sem` 被另一次 `sem_init()` 调用重新初始化之前，后续使用信号量 `sem` 的效果是未定义的。

在没有线程当前被阻塞的已初始化信号量上销毁它是安全的。在有其他线程当前被阻塞的信号量上销毁的效果是未定义的。

返回值

成功完成后，应返回零值。否则，应返回 -1 值并设置 `errno` 来指示错误。

错误

`sem_destroy()` 函数可能失败的情况：

- **[EINVAL]**

`sem` 参数不是有效的信号量。

- **[EBUSY]**

当前有进程在该信号量上阻塞。

示例

无。

应用用法

无。

基本原理

无。

未来方向

无。

参见

- `semctl()`
- `semget()`
- `semop()`
- `sem_init()`
- `sem_open()`

XBD `<semaphore.h>`

变更历史

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

- `sem_destroy()` 函数被标记为信号量选项的一部分。
- 如果实现不支持信号量选项，不需要提供存根，因此 `[ENOSYS]` 错误条件已被移除。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/114，更新了错误部分，使 `[EINVAL]` 错误变为可选的。

Issue 7

- `sem_destroy()` 函数从信号量选项移至基本规范。
- 应用了 POSIX.1-2008, 技术勘误 1, XSH/TC1-2008/0524 [37]。

Issue 8

- 应用了 Austin Group 缺陷 368, 添加了一个要求: 如果无名信号量使用文件描述符实现, `sem_destroy()` 会关闭文件描述符。
-

1.193. sem_getvalue

SYNOPSIS

```
#include <semaphore.h>

int sem_getvalue(sem_t *restrict sem, int *restrict sval);
```

DESCRIPTION

`sem_getvalue()` 函数应将 `sval` 参数引用的位置更新为 `sem` 引用的信号量的值，而不影响信号量的状态。更新后的值表示在调用期间某个未指定时刻发生实际信号量值，但不一定是返回给调用进程时的实际信号量值。

如果 `sem` 被锁定，那么 `sval` 指向的对象要么被设置为零，要么被设置为一个负数，其绝对值表示在调用期间某个未指定时刻等待信号量的进程数。

RETURN VALUE

成功完成后，`sem_getvalue()` 函数应返回零值。否则，它应返回 -1 值并设置 `errno` 来指示错误。

ERRORS

`sem_getvalue()` 函数可能会失败，原因如下：

- **[EINVAL]** —— `sem` 参数不引用有效的信号量。

以下为补充信息部分。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`
- `sem_trywait()`

XBD `<semaphore.h>`

CHANGE HISTORY

首次发布于 Issue 5。

为了与 POSIX 实时扩展对齐而包含。

Issue 6

`sem_getvalue()` 函数被标记为信号量选项的一部分。

如果实现不支持信号量选项，则不需要提供存根，因此 [ENOSYS] 错误条件已被移除。

为了与 IEEE Std 1003.1d-1999 对齐，`sem_timedwait()` 函数被添加到 SEE ALSO 部分。

为了与 ISO/IEC 9899:1999 标准对齐，`sem_getvalue()` 原型中添加了 **restrict** 关键字。

应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/54。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/115，更新了 ERRORS 部分，使得 [EINVAL] 错误变为可选的。

Issue 7

`sem_getvalue()` 函数从信号量选项移动到 Base。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0525 [37]。

1.194. sem_init — 初始化匿名信号量

概要

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned value);
```

描述

`sem_init()` 函数应当初始化由 `sem` 引用的匿名信号量。初始化后的信号量值应当为 `value`。在成功调用 `sem_init()` 之后，该信号量可在后续的 `sem_clockwait()`、`sem_destroy()`、`sem_post()`、`sem_timedwait()`、`sem_trywait()` 和 `sem_wait()` 调用中使用。此信号量在被销毁前应保持可用状态。匿名信号量可使用文件描述符实现。

如果 `pshared` 参数具有非零值，则信号量在进程间共享；在这种情况下，任何可以访问信号量 `sem` 的进程都可以使用 `sem` 执行 `sem_clockwait()`、`sem_destroy()`、`sem_post()`、`sem_timedwait()`、`sem_trywait()` 和 `sem_wait()` 操作。

如果 `pshared` 参数为零，则信号量在同一进程的线程间共享；此进程中的任何线程都可以使用 `sem` 执行 `sem_clockwait()`、`sem_destroy()`、`sem_post()`、`sem_timedwait()`、`sem_trywait()` 和 `sem_wait()` 操作。

关于进一步要求，请参见 2.9.9 同步对象副本和替代映射。

尝试初始化已经初始化的信号量会导致未定义行为。

返回值

成功完成时，`sem_init()` 函数应当初始化 `sem` 中的信号量并返回 0。否则，它应当返回 -1 并设置 `errno` 来指示错误。

错误

`sem_init()` 函数在以下情况下应当失败：

- **[EINVAL]**

`value` 参数超过 {SEM_VALUE_MAX}。

- **[ENOSPC]**

初始化信号量所需的资源已耗尽，或者已达到信号量限制 ({SEM_NSEMS_MAX})。

- **[EPERM]**

进程缺乏初始化信号量的适当权限。

`sem_init()` 函数在以下情况下可能会失败：

- **[EMFILE]**

进程可用的所有文件描述符当前都已打开。

- **[ENFILE]**

系统中当前打开的文件数量已达到最大允许数量。

示例

无。

应用程序使用

无。

基本原理

无。

未来方向

无。

另请参见

- `sem_clockwait()`

- `sem_destroy()`

- `sem_post()`
- `sem_trywait()`
- `<semaphore.h>`

更改历史

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

- `sem_init()` 函数被标记为信号量选项的一部分。
- 如果实现不支持信号量选项，则不需要提供存根，因此移除了 [ENOSYS] 错误条件。
- 为与 IEEE Std 1003.1d-1999 对齐，在另请参见部分添加了 `sem_timedwait()` 函数。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/116，更新描述以添加 `sem_timedwait()` 函数，以与 IEEE Std 1003.1d-1999 对齐。

Issue 7

- 应用了 SD5-XSH-ERN-176。
- `sem_init()` 函数从信号量选项移动到基础规范。
- 应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0526 [37]。
- 应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0318 [972]。

Issue 8

- 应用了 Austin Group 缺陷 368，添加了匿名信号量可使用文件描述符实现的声明，并添加了 [EMFILE] 和 [ENFILE] 错误。
- 应用了 Austin Group 缺陷 1216，添加了 `sem_clockwait()`。

1.195. sem_open — 初始化并打开命名信号量

概要

```
#include <semaphore.h>

sem_t *sem_open(const char *name, int oflag, ...);
```

描述

`sem_open()` 函数应在命名信号量和进程之间建立连接。命名信号量可以使用文件描述符实现。在以信号量名称 `name` 调用 `sem_open()` 后，进程可以使用该调用返回的信号量句柄引用与 `name` 关联的信号量。此信号量可用于后续调用 `sem_clockwait()`、`sem_close()`、`sem_post()`、`sem_timedwait()`、`sem_trywait()` 和 `sem_wait()`。该信号量保持对此进程可用，直到通过成功调用 `sem_close()`、`_exit()` 或 `exec` 函数之一关闭信号量。

`oflag` 参数控制信号量是被创建还是仅被 `sem_open()` 调用访问。以下标志位可以在 `oflag` 中设置：

O_CREAT

如果信号量尚不存在，此标志用于创建信号量。如果设置了 O_CREAT 且信号量已存在，则 O_CREAT 无效，O_EXCL 下注明的除外。否则，`sem_open()` 创建命名信号量。O_CREAT 标志需要第三个和第四个参数：`mode`（类型为 `mode_t`）和 `value`（类型为 `unsigned`）。信号量以 `value` 作为初始值创建。信号量的有效初始值小于或等于 {SEM_VALUE_MAX}。

信号量的用户 ID 应设置为进程的有效用户 ID。信号量的组 ID 应设置为进程的有效组 ID；但是，如果 `name` 参数在文件系统中可见，组 ID 可以设置为包含目录的组 ID。信号量的权限位设置为 `mode` 参数的值，进程文件模式创建掩码中设置的位除外。当指定了文件权限位之外的 `mode` 位时，效果是未指定的。

在使用 O_CREAT 标志通过 `sem_open()` 创建名为 `name` 的信号量后，其他进程可以通过调用 `sem_open()` 并使用相同的 `name` 值连接到该信号量。

O_EXCL

如果设置了 O_EXCL 和 O_CREAT，且信号量 `name` 已存在，则 `sem_open()` 失败。对于其他执行带有 O_EXCL 和 O_CREAT 设置的 `sem_open()` 的进程，检查信号量是否存在以及信号量不存在时创建信号量的操作是原子的。如果设置了 O_EXCL 但未设置 O_CREAT，则效果是未定义的。

如果在 `oflag` 参数中指定了 O_CREAT 和 O_EXCL 之外的标志，则效果是未指定的。

`name` 参数指向命名信号量对象的字符串。名称是否出现在文件系统中并对以路径名为参数的函数可见是未指定的。`name` 参数符合路径名的构造规则，但 `name` 中除前导 `<slash>` 字符外的 `<slash>` 字符的解释是实现定义的，并且 `name` 参数的长度限制是实现定义的，不必与路径名限制 {PATH_MAX} 和 {NAME_MAX} 相同。如果 `name` 以 `<slash>` 字符开头，则调用 `sem_open()` 且具有相同 `name` 值的进程应引用同一信号量对象，只要该名称尚未被删除。如果 `name` 不以 `<slash>` 字符开头，则效果是实现定义的。

如果一个进程对相同的 `name` 值进行了多次成功的 `sem_open()` 调用，期间没有对 `name` 进行 `sem_unlink()` 调用，并且该信号量的至少一个打开句柄尚未通过 `sem_close()` 调用关闭，则对于每次此类成功调用是返回相同句柄还是唯一句柄是实现定义的。

对信号量副本的引用会产生未定义结果。

返回值

成功完成后，`sem_open()` 函数应返回信号量的地址。否则，应返回 SEM_FAILED 值并设置 `errno` 以指示错误。符号 SEM_FAILED 在 `<semaphore.h>` 头文件中定义。`sem_open()` 的成功返回不应返回 SEM_FAILED 值。

错误

如果出现以下任何条件，`sem_open()` 函数应返回 SEM_FAILED 并将 `errno` 设置为相应的值：

- [EACCES] 命名信号量存在且 `oflag` 指定的权限被拒绝，或者命名信号量不存在且创建命名信号量的权限被拒绝。
- [EEXIST] 设置了 O_CREAT 和 O_EXCL 且命名信号量已存在。
- [EINTR] `sem_open()` 操作被信号中断。

- **[EINVAL]** 给定名称不支持 `sem_open()` 操作，或者在 `oflag` 中指定了 `O_CREAT` 且 `value` 大于 `{SEM_VALUE_MAX}`。
- **[ENOENT]** 未设置 `O_CREAT` 且命名信号量不存在。
- **[ENOMEM]** 创建新命名信号量的内存不足。
- **[ENOSPC]** 存储设备上没有足够的空间来创建新命名信号量。

如果出现以下任何条件，`sem_open()` 函数可能返回 `SEM_FAILED` 并将 `errno` 设置为相应的值：

- **[EMFILE]** 此进程当前使用的信号量描述符或文件描述符过多。
- **[ENAMETOOLONG]** 在不支持 XSI 选项 [XSI] 的系统上，`name` 参数的长度超过 `{_POSIX_PATH_MAX}`，在 XSI 系统上超过 `{_XOPEN_PATH_MAX}`，或者在非 XSI 系统上路径名组件长度超过 `{_POSIX_NAME_MAX}`，在 XSI 系统上超过 `{_XOPEN_NAME_MAX}`。
- **[ENFILE]** 系统中当前打开的信号量描述符或文件描述符过多。

示例

无。

应用程序用法

无。

原理

早期草案要求 `sem_open()` 函数返回类型为 `sem_t *` 的错误返回值 `-1`，这不能保证在实现之间可移植。修订后的文本提供符号错误代码 `SEM_FAILED` 以消除类型冲突。

未来方向

未来版本可能要求 `sem_open()` 和 `sem_unlink()` 函数具有类似于普通文件系统操作的语义。

另请参阅

`semctl()` 、 `semget()` 、 `semop()` 、 `sem_clockwait()` 、
`sem_close()` 、 `sem_post()` 、 `sem_trywait()` 、 `sem_unlink()`

XBD `<semaphore.h>`

更改历史

首次在 Issue 5 中发布。包含用于与 POSIX 实时扩展对齐。

Issue 6

`sem_open()` 函数被标记为信号量选项的一部分。

[ENOSYS] 错误条件已被移除，因为如果实现不支持信号量选项，则不需要提供存根。

`sem_timedwait()` 函数已添加到另请参阅部分，以与 IEEE Std 1003.1d-1999 对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/117，更新描述以添加 `sem_timedwait()` 函数，以与 IEEE Std 1003.1d-1999 对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/118，更新描述以描述在调用 `sem_open()` 时返回相同信号量地址的条件。添加了"并且至少一次先前的成功 `sem_open()` 调用未与 `sem_close()` 调用匹配"的文字。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #066，更新了 [ENOSPC] 错误情况并添加了 [ENOMEM] 错误情况。

应用了 Austin Group Interpretation 1003.1-2001 #077，阐明了 `name` 参数并添加了 [ENAMETOOLONG] 作为"可能失败"错误。

应用了 Austin Group Interpretation 1003.1-2001 #141，添加了未来方向。

应用了 SD5-XSH-ERN-170，更新描述以阐明设置信号量用户 ID 和组 ID 的措辞。

`sem_open()` 函数从信号量选项移动到基础。

应用了 POSIX.1-2008， Technical Corrigendum 1， XSH/TC1-2008/0527 [37]。

Issue 8

应用了 Austin Group Defect 368，添加了命名信号量可以使用文件描述符实现的语句并更改了错误部分。

应用了 Austin Group Defect 1216，添加了 `sem_clockwait()`。

应用了 Austin Group Defect 1324，当对相同的 `name` 值进行多次成功的 `sem_open()` 调用时，是返回相同句柄还是唯一句柄成为实现定义的。

1.196. sem_post — 解锁信号量

概要

```
#include <semaphore.h>

int sem_post(sem_t *sem);
```

描述

`sem_post()` 函数应通过在由 `sem` 引用的信号量上执行信号量解锁操作来解锁该信号量。

如果此操作产生的信号量值为正值，则没有线程被阻塞等待信号量解锁；信号量值仅被简单地递增。

如果此操作产生的信号量值为零，则应允许其中一个被阻塞等待信号量的线程从其对 `sem_wait()` 的调用中成功返回。[PS] 如果支持进程调度选项，则应选择适合被阻塞线程生效的调度策略和参数的方式来选择要解除阻塞的线程。对于 SCHED_FIFO 和 SCHED_RR 调度器，应解除最高优先级等待线程的阻塞，如果有多个最高优先级线程被阻塞等待信号量，则应解除等待时间最长的最高优先级线程的阻塞。如果未定义进程调度选项，则选择要解除阻塞的线程是未指定的。

[SS] 如果支持进程零星服务器选项，且调度策略为 SCHED_SPORADIC，则语义与上述 SCHED_FIFO 相同。

`sem_post()` 函数应为异步信号安全函数，可以从信号捕获函数中调用。

返回值

如果成功，`sem_post()` 函数应返回零；否则，函数应返回 -1 并设置 `errno` 以指示错误。

错误

`sem_post()` 函数在以下情况下应失败：

- [EOVERFLOW] 将超过信号量的最大允许值。

`sem_post()` 函数在以下情况下可能失败：

- [EINVAL] `sem` 参数不引用有效的信号量。

示例

参见 `sem_clockwait()`。

应用用法

无。

原理

无。

未来方向

无。

参见

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_trywait()`

XBD 4.15.2 内存同步, `<semaphore.h>`

变更历史

首次发布于 Issue 5

为与 POSIX 实时扩展对齐而包含。

Issue 6

- `sem_post()` 函数被标记为信号量选项的一部分。
- 删除了 [ENOSYS] 错误条件，因为如果实现不支持信号量选项，则无需提供存根。
- 为与 IEEE Std 1003.1d-1999 对齐，在参见部分添加了 `sem_timedwait()` 函数。
- 为与 IEEE Std 1003.1d-1999 对齐，在指定要解除阻塞线程的调度策略列表中添加了 SCHED_SPORADIC。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/119，更新错误部分，使 [EINVAL] 错误变为可选。

Issue 7

- 应用了 Austin Group 解释 1003.1-2001 #156。
- `sem_post()` 函数从信号量选项移至基础部分。
- 应用了 POSIX.1-2008，技术勘误表 1，XSH/TC1-2008/0528 [37]。

Issue 8

- 应用了 Austin Group 缺陷 315，添加了 [EOVERFLOW] 错误。
-

1.197. sem_clockwait, sem_timedwait — 锁定信号量

概要

```
#include <semaphore.h>

int sem_clockwait(sem_t *restrict sem,
                  clockid_t clock_id,
                  const struct timespec *restrict abstime);

int sem_timedwait(sem_t *restrict sem,
                  const struct timespec *restrict abstime);
```

描述

`sem_clockwait()` 和 `sem_timedwait()` 函数应锁定 `sem` 引用的信号量，如同 `sem_wait()` 函数一样。但是，如果信号量无法在不等待另一个进程或线程通过执行 `sem_post()` 函数来解锁信号量的情况下被锁定，则当指定的超时时间到期时，此等待应被终止。

当 `abstime` 指定的绝对时间经过时，超时应到期，这是基于超时的时钟测量的（即，当时钟值等于或超过 `abstime` 时），或者如果在调用时 `abstime` 指定的绝对时间已经过去。

对于 `sem_timedwait()`，超时应基于 `CLOCK_REALTIME` 时钟。对于 `sem_clockwait()`，超时应基于 `clock_id` 参数指定的时钟。超时的分辨率应是其所基于时钟的分辨率。实现应支持将 `CLOCK_REALTIME` 和 `CLOCK_MONOTONIC` 作为 `clock_id` 参数传递给 `sem_clockwait()`。

在任何情况下，如果信号量可以立即锁定，函数不应因超时而失败。如果信号量可以立即锁定，则无需检查 `abstime` 的有效性。

返回值

如果调用进程成功在指定信号量 `sem` 上执行了信号量锁定操作，`sem_clockwait()` 和 `sem_timedwait()` 函数应返回零。如果调用不成功，信号量的状态应保持不变，函数应返回值 -1 并设置 `errno` 来指示错误。

错误

`sem_clockwait()` 和 `sem_timedwait()` 函数在以下情况下应失败：

- **[EINVAL]**
- 进程或线程将会阻塞，且 `abstime` 参数指定的纳秒字段值小于零或大于等于 1000 百万，或者 `sem_clockwait()` 函数传递了无效或不支持的 `clock_id` 值。
- **[ETIMEDOUT]**
- 在指定的超时时间到期之前无法锁定信号量。

`sem_clockwait()` 和 `sem_timedwait()` 函数在以下情况下可能失败：

- **[EDEADLK]**
- 检测到死锁条件。
- **[EINTR]**
- 信号中断了函数。
- **[EINVAL]**
- `sem` 参数不引用有效的信号量。

示例

下面显示的程序操作一个未命名信号量。程序期望两个命令行参数。第一个参数指定一个秒值，用于设置闹钟定时器来生成 SIGALRM 信号。此处理程序执行 `sem_post()` 来增加在 `main()` 中使用 `sem_clockwait()` 等待的信号量。第二个命令行参数指定 `sem_clockwait()` 的超时长度，以秒为单位。

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <semaphore.h>
#include <time.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>

sem_t sem;

static void
handler(int sig)
```

```
{  
    int sav_errno = errno;  
    static const char info_msg[] = "sem_post() from handler\n";  
    write(STDOUT_FILENO, info_msg, sizeof info_msg - 1);  
    if (sem_post(&sem) == -1) {  
        static const char err_msg[] = "sem_post() failed\n";  
        write(STDERR_FILENO, err_msg, sizeof err_msg - 1);  
        _exit(EXIT_FAILURE);  
    }  
    errno = sav_errno;  
}  
  
int  
main(int argc, char *argv[])  
{  
    struct sigaction sa;  
    struct timespec ts;  
    int s;  
  
    if (argc != 3) {  
        fprintf(stderr, "Usage: %s <alarm-secs> <wait-secs>\n",  
                argv[0]);  
        exit(EXIT_FAILURE);  
    }  
  
    if (sem_init(&sem, 0, 0) == -1) {  
        perror("sem_init");  
        exit(EXIT_FAILURE);  
    }  
  
    /* 建立 SIGALRM 处理程序; 使用 argv[1] 设置闹钟定时器 */  
  
    sa.sa_handler = handler;  
    sigemptyset(&sa.sa_mask);  
    sa.sa_flags = 0;  
    if (sigaction(SIGALRM, &sa, NULL) == -1) {  
        perror("sigaction");  
        exit(EXIT_FAILURE);  
    }  
  
    alarm(atoi(argv[1]));  
  
    /* 计算相对间隔为当前时间加上 argv[2] 给定的秒数 */  
  
    if (clock_gettime(CLOCK_MONOTONIC, &ts) == -1) {  
        perror("clock_gettime");  
        exit(EXIT_FAILURE);  
    }  
    ts.tv_sec += atoi(argv[2]);
```

```
printf("main() about to call sem_clockwait()\n");
while ((s = sem_clockwait(&sem, CLOCK_MONOTONIC, &ts)) == -1
       &amp; errno == EINTR)
    continue; /* 如果被处理程序中断则重启 */

/* 检查发生了什么 */

if (s == -1) {
    if (errno == ETIMEDOUT)
        printf("sem_clockwait() timed out\n");
    else
        perror("sem_clockwait");
} else
    printf("sem_clockwait() succeeded\n");

exit((s == 0) ? EXIT_SUCCESS : EXIT_FAILURE);
}
```

应用程序用法

使用这些函数的应用程序可能会受到优先级反转的影响，如 XBD 3.275 优先级反转中所述。

原理

无。

未来方向

无。

另请参阅

- [sem_post\(\)](#)
- [sem_trywait\(\)](#)
- [semctl\(\)](#)
- [semget\(\)](#)
- [semop\(\)](#)

- `time()`

XBD 3.275 优先级反转, `<semaphore.h>`, `<time.h>`

变更历史

首次发布于 Issue 6。源自 IEEE Std 1003.1d-1999。

应用了 IEEE Std 1003.1-2001/Cor 2-2004, 项目 XSH/TC2/D6/120, 更新了 ERRORS 部分, 使得 [EINVAL] 错误变为可选的。

Issue 7

`sem_timedwait()` 函数从信号量选项移至基础部分。

与定时器选项相关的功能移至基础部分。

添加了一个示例。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0529 [138]。

Issue 8

应用了 Austin Group Defect 592, 从 SYNOPSIS 和 DESCRIPTION 部分移除了与 `<time.h>` 相关的文本。

应用了 Austin Group Defect 1216, 添加了 `sem_clockwait()`。

1.198. sem_trywait, sem_wait — 锁定信号量

SYNOPSIS (概要)

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

DESCRIPTION (描述)

`sem_trywait()` 函数仅在信号量当前未被锁定时才锁定 `sem` 所引用的信号量；也就是说，仅当信号量值当前为正数时才锁定。否则，该函数不应锁定该信号量。

`sem_wait()` 函数通过在该信号量上执行信号量锁定操作来锁定 `sem` 所引用的信号量。如果信号量值当前为零，那么调用线程在锁定该信号量或调用被信号中断之前，不应从 `sem_wait()` 调用中返回。

成功返回时，信号量的状态应为已锁定，并且应保持锁定状态，直到 `sem_post()` 函数被执行并成功返回。

`sem_wait()` 函数可被信号传递中断。

RETURN VALUE (返回值)

如果调用进程在 `sem` 指定的信号量上成功执行了信号量锁定操作，`sem_trywait()` 和 `sem_wait()` 函数应返回零。如果调用不成功，信号量的状态应保持不变，函数应返回值 -1 并设置 `errno` 以指示错误。

ERRORS (错误)

`sem_trywait()` 函数在以下情况下应失败：

- **[EAGAIN]**

信号量已被锁定，因此无法通过 `sem_trywait()` 操作立即锁定。

`sem_trywait()` 和 `sem_wait()` 函数在以下情况下可能失败：

- **[EDEADLK]**

检测到死锁条件。

- **[EINTR]**

信号中断了此函数。

- **[EINVAL]**

`sem` 参数不引用有效的信号量。

APPLICATION USAGE (应用程序使用)

使用这些函数的应用程序可能会受到优先级反转的影响，如 XBD 3.275 优先级反转中所述。

SEE ALSO (另请参见)

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`

XBD:

- 3.275 优先级反转
- 4.15.2 内存同步
-

CHANGE HISTORY (变更历史)

Issue 6

- `sem_trywait()` 和 `sem_wait()` 函数被标记为信号量选项的一部分。
- 如果实现不支持信号量选项，则不需要提供存根，因此移除了 [ENOSYS] 错误条件。
- 为了与 IEEE Std 1003.1d-1999 对齐，在另请参见部分添加了 `sem_timedwait()` 函数。

- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/121，更新了错误部分，使 [EINVAL] 错误变为可选的。

Issue 7

- 应用了 SD5-XSH-ERN-54，从"应失败"错误情况中移除了 `sem_wait()` 函数。
 - `sem_trywait()` 和 `sem_wait()` 函数从信号量选项移动到基本定义。
 - 应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0530 [37]。
-

1.199. sem_unlink - 删 除 命 名 信 号 量

概要

```
#include <semaphore.h>

int sem_unlink(const char *name);
```

描述

`sem_unlink()` 函数应删除由字符串 `name` 命名的信号量。如果名为 `name` 的信号量当前被其他进程引用，那么 `sem_unlink()` 对信号量的状态不应产生影响。如果在调用 `sem_unlink()` 时有一个或多个进程打开该信号量，则信号量的销毁将被推迟，直到所有对信号量的引用都通过调用 `sem_close()`、`_exit()` 或 `exec` 被销毁。在调用 `sem_unlink()` 后，调用 `sem_open()` 重新创建或重新连接信号量将引用一个新的信号量。`sem_unlink()` 调用不会阻塞直到所有引用都被销毁；它应立即返回。

返 回 值

成功完成时，`sem_unlink()` 函数应返回值 0。否则，信号量不应改变，函数应返回值 -1 并设置 `errno` 以指示错误。

错 误

在以下情况下，`sem_unlink()` 函数应失败：

- **[EACCES]** - 拒绝解除链接命名信号量的权限。
- **[ENOENT]** - 命名信号量不存在。

在以下情况下，`sem_unlink()` 函数可能失败：

- **[ENAMETOOLONG]** - `name` 参数的长度在不支持 XSI 选项的系统上超过 `{_POSIX_PATH_MAX}`，在 XSI 系统上超过 `{_XOPEN_PATH_MAX}`，或者包含的路径名组件在不支持 XSI 选项的系统上长于 `{_POSIX_NAME_MAX}`，在 XSI 系统上长于 `{_XOPEN_NAME_MAX}`。使用包含与先前成功的

`sem_open()` 调用中使用的相同信号量名称的 `name` 参数调用 `sem_unlink()` 不应产生 [ENAMETOOLONG] 错误。

示例

无。

应用用法

无。

基本原理

无。

未来方向

未来版本可能要求 `sem_open()` 和 `sem_unlink()` 函数具有类似于普通文件系统操作的语义。

另请参阅

- `semctl()`
- `semget()`
- `semop()`
- `sem_close()`
- `sem_open()`
- XBD `<semaphore.h>`

变更历史

第 5 版首次发布

为了与 POSIX 实时扩展保持一致而包含。

第 6 版

- `sem_unlink()` 函数被标记为信号量选项的一部分。
- [ENOSYS] 错误条件已被删除，因为如果实现不支持信号量选项，则无需提供存根。

第 7 版

- 应用 Austin Group 解释 1003.1-2001 #077，将 [ENAMETOOLONG] 从"应失败"错误改为"可能失败"错误。
 - 应用 Austin Group 解释 1003.1-2001 #141，添加未来方向。
 - `sem_unlink()` 函数从信号量选项移至基础部分。
 - 应用 POSIX.1-2008，技术勘误表 1，XSH/TC1-2008/0531 [37]。
-

1.200. sem_trywait, sem_wait — 锁定信号量

概要

```
#include <semaphore.h>

int sem_trywait(sem_t *sem);
int sem_wait(sem_t *sem);
```

描述

`sem_trywait()` 函数只有在信号量当前未被锁定时才锁定由 `sem` 引用的信号量；也就是说，只有当信号量当前值为正数时才锁定。否则，该函数不会锁定信号量。

`sem_wait()` 函数通过对该信号量执行信号量锁定操作来锁定由 `sem` 引用的信号量。如果信号量当前值为零，那么调用线程在能够锁定信号量或调用被信号中断之前，不会从 `sem_wait()` 调用中返回。

成功返回时，信号量的状态应被锁定，并将保持锁定状态，直到 `sem_post()` 函数被执行并成功返回。

`sem_wait()` 函数可被信号的传送中断。

返回值

如果调用进程成功对指定信号量 `sem` 执行了信号量锁定操作，`sem_trywait()` 和 `sem_wait()` 函数应返回零。如果调用不成功，信号量的状态应保持不变，函数应返回值 -1，并设置 `errno` 来指示错误。

错误

`sem_trywait()` 函数在以下情况下应失败：

- **[EAGAIN]**

信号量已被锁定，因此无法通过 `sem_trywait()` 操作立即锁定。

`sem_trywait()` 和 `sem_wait()` 函数在以下情况下可能失败：

- **[EDEADLK]**

检测到死锁条件。

- **[EINTR]**

信号中断了此函数。

- **[EINVAL]**

`sem` 参数不引用有效的信号量。

应用程序使用

使用这些函数的应用程序可能会受到优先级反转的影响，如 XBD 3.275 优先级反转中所述。

原理

无。

未来方向

无。

参见

- `semctl()`
- `semget()`
- `semop()`
- `sem_clockwait()`
- `sem_post()`

XBD 3.275 优先级反转，4.15.2 内存同步，`<semaphore.h>`

变更历史

首次发布于 Issue 5

包含以与 POSIX 实时扩展对齐。

Issue 6

- `sem_trywait()` 和 `sem_wait()` 函数被标记为信号量选项的一部分。
- [ENOSYS] 错误条件已被移除，因为如果实现不支持信号量选项，则无需提供存根。
- `sem_timedwait()` 函数被添加到参见部分，以与 IEEE Std 1003.1d-1999 对齐。
- 应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/121，更新错误部分，使 [EINVAL] 错误变为可选。

Issue 7

- 应用 SD5-XSH-ERN-54，将 `sem_wait()` 函数从"应失败"错误情况中移除。
 - `sem_trywait()` 和 `sem_wait()` 函数从信号量选项移至基础部分。
 - 应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0530 [37]。
-

1.201. setbuf

SYNOPSIS (概要)

```
#include <stdio.h>

void setbuf(FILE *restrict stream, char *restrict buf);
```

DESCRIPTION (描述)

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

除了不返回值外，函数调用：

```
setbuf(stream, buf)
```

应等效于：

```
setvbuf(stream, buf, _IOFBF, BUFSIZ)
```

如果 `buf` 不是空指针，或者等效于：

```
setvbuf(stream, buf, _IONBF, BUFSIZ)
```

如果 `buf` 是空指针。

RETURN VALUE (返回值)

`setbuf()` 函数不应返回任何值。

ERRORS (错误)

尽管 `setvbuf()` 接口可能以定义的方式设置 `errno`，但在调用 `setbuf()` 后 `errno` 的值是未指定的。

以下各节为提供参考信息的部分。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

一个常见的错误源是在代码块中将缓冲区空间分配为"自动"变量，然后未能在同一代码块中关闭流。

使用 `setbuf()` 时，分配 BUFSIZ 字节的缓冲区不一定意味着所有 BUFSIZ 字节都用于缓冲区。

由于成功时不要求 `errno` 保持不变，为了正确检测并可能从错误中恢复，应用程序应使用 `setvbuf()` 而不是 `setbuf()`。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- [2.5 标准I/O流](#)
- [fopen\(\)](#)
- [setvbuf\(\)](#)

XBD

CHANGE HISTORY (变更历史)

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 6

`setbuf()` 的原型已更新以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

应用了 POSIX.1-2008、Technical Corrigendum 1、XSH/TC1-2008/0546 [397]、XSH/TC1-2008/0547 [397] 和 XSH/TC1-2008/0548 [14]。

1.202. setenv — 添加或更改环境变量

概要 (SYNOPSIS)

```
#include <stdlib.h>

int setenv(const char *envname, const char *envval, int overwri
```

描述 (DESCRIPTION)

`setenv()` 函数应在调用进程的环境中更新或添加变量。`envname` 参数指向一个字符串，该字符串包含要添加或修改的环境变量的名称。环境变量应设置为 `envval` 所指向的值。如果 `envname` 指向的字符串包含 '=' 字符，则函数应失败。

如果 `envname` 指定的环境变量已经存在且 `overwrite` 的值非零，则函数应返回成功并更新环境。如果 `envname` 指定的环境变量已经存在且 `overwrite` 的值为零，则函数应返回成功且环境保持不变。

`setenv()` 函数应更新 `environ` 所指向的指针列表。

`envname` 和 `envval` 描述的字符串会被此函数复制。

`setenv()` 函数不必是线程安全的。

返回值 (RETURN VALUE)

成功完成后，应返回零。否则，应返回 -1，并设置 `errno` 以指示错误，且环境应保持不变。

错误 (ERRORS)

`setenv()` 函数应在以下情况下失败：

- **[EINVAL]**
- `envname` 参数指向空字符串或指向包含 '=' 字符的字符串。
- **[ENOMEM]**

- 没有足够的可用内存来将变量或其值添加到环境中。

示例 (EXAMPLES)

无。

应用程序使用 (APPLICATION USAGE)

关于在多线程应用程序中更改环境的限制，请参见 [exec\(\)](#)。

基本原理 (RATIONALE)

如果 `setenv()` 更改外部变量 `environ`，可能会发生预期之外的结果。特别是，如果 `main()` 的可选 `envp` 参数存在，它不会被更改，因此可能指向环境的过时副本（`environ` 的任何其他副本也是如此）。然而，除了上述限制之外，标准开发者希望支持通过 `environ` 指针遍历环境的传统方法。

标准开发者决定在此版本中要求 `setenv()`，因为它解决了一个缺失的功能，并且不会对实现者造成重大负担。

关于是否应该将 System V 的 `putenv()` 函数或 BSD 的 `setenv()` 函数作为必需函数存在相当大的争议。最终选择了 `setenv()` 函数，因为它允许实现 `unsetenv()` 函数来删除环境变量，而无需指定额外的接口。`putenv()` 函数作为 XSI 选项的一部分可用。

标准开发者曾考虑要求当调用 `setenv()` 会导致超过 `{ARG_MAX}` 时指示错误。该要求被拒绝，因为这种条件可能是暂时的，应用程序最终可能会减少环境大小。最终的成功或失败取决于调用 `exec` 时的大小，`exec()` 会返回此错误条件的指示。

另参见 `getenv()` 的基本原理部分。

未来方向 (FUTURE DIRECTIONS)

无。

另见 (SEE ALSO)

[exec](#),
[getenv\(\)](#),
[putenv\(\)](#),
[unsetenv\(\)](#)

XBD ,
,

变更历史 (CHANGE HISTORY)

首次发布于 Issue 6。源自 IEEE P1003.1a 草案标准。

IEEE Std 1003.1-2001/Cor 1-2002, 项目 XSH/TC1/D6/55 被应用, 在应用程序使用和另见部分添加了对 [exec](#) 的引用。

Issue 7

Austin Group Interpretation 1003.1-2001 #156 被应用。

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0549 [167], XSH/TC1-2008/0550 [185], XSH/TC1-2008/0551 [167], 和 XSH/TC1-2008/0552 [38] 被应用。

1.203. setjmp

概要

```
#include <setjmp.h>
void setjmp(jmp_buf env);
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

对 `setjmp()` 的调用应将其调用环境保存在其 `env` 参数中，以供 `longjmp()` 稍后使用。

`setjmp()` 是宏还是函数是未指定的。如果为了访问实际函数而抑制宏定义，或者程序定义了名为 `setjmp` 的外部标识符，则行为是未定义的。

应用程序应确保 `setjmp()` 的调用仅出现在以下上下文中：

- 选择或迭代语句的整个控制表达式
- 关系或相等运算符的一个操作数，另一个操作数为整型常量表达式，结果表达式为选择或迭代语句的整个控制表达式
- 一元 '!' 运算符的操作数，结果表达式为选择或迭代语句的整个控制表达式
- 表达式语句的整个表达式（可能强制转换为 `void`）

如果调用出现在任何其他上下文中，则行为是未定义的。

以下部分是提供信息的。

信息性文本结束。

1.204. setlocale — 设置程序本地化环境

概要

```
#include <locale.h>

char *setlocale(int category, const char *locale);
```

描述

`setlocale()` 函数根据 `category` 和 `locale` 参数选择全局本地化环境的适当部分，可用于更改或查询整个全局本地化环境或其部分内容。对于 `category`，值 `LC_ALL` 表示整个全局本地化环境；其他值仅表示全局本地化环境的一部分：

- **LC_COLLATE**

影响正则表达式和排序函数的行为。

- **LC_CTYPE**

影响正则表达式、字符分类、字符转换函数和宽字符函数的行为。

- **LC_MESSAGES**

影响 `nl_langinfo()` 返回的肯定和否定响应表达式以及消息目录的定位方式。它也可能影响返回或写入消息字符串的函数的行为。

- **LC_MONETARY**

影响处理货币值的函数的行为。

- **LC_NUMERIC**

影响处理数值的函数的行为。

- **LC_TIME**

影响时间转换函数的行为。

`locale` 参数是一个指向字符串的指针，该字符串包含 `category` 的所需设置。此字符串的内容由实现定义。此外，为 `category` 的所有设置定义了以下预设的 `locale` 值：

- **"POSIX"**

指定用于 C 语言翻译的最小环境，称为 POSIX 本地化环境。POSIX 本地化环境是在进入 `main()` 时的默认全局本地化环境。

- "C"

等同于 "POSIX"。

- ""

指定由实现定义的本机环境。指定类别的新本地化环境名称的确定取决于相关环境变量 `LC_*` 和 `LANG` 的值；参见 POSIX.1-2024 基础定义卷。

- 空指针

指示 `setlocale()` 查询当前全局本地化环境设置，如果 `category` 不是 `LC_ALL`，则返回本地化环境的名称；如果 `category` 是 `LC_ALL`，则返回一个编码所有单个类别本地化环境名称的字符串。

设置全局本地化环境的所有类别类似于连续设置全局本地化环境的每个单独类别，不同之处在于所有错误检查都在执行任何操作之前完成。要设置全局本地化环境的所有类别，可以按以下方式调用 `setlocale()`：

```
setlocale(LC_ALL, "");
```

在这种情况下，`setlocale()` 应首先根据优先级规则验证它需要的所有环境变量的值是否表示支持的本地化环境。如果这些环境变量搜索中的任何一个产生的本地化环境不受支持（且非空），`setlocale()` 应返回空指针且全局本地化环境不应更改。如果所有环境变量都命名了支持的本地化环境，`setlocale()` 应继续执行，就好像已为每个类别调用过它一样，使用相关环境变量的适当值，如果没有这样的值，则使用实现定义的默认值。

使用 `setlocale()` 建立的全局本地化环境仅应用于那些没有使用 `uselocale()` 设置当前本地化环境的线程，或者其当前本地化环境已使用 `uselocale(LC_GLOBAL_LOCALE)` 设置为全局本地化环境的线程。

实现的行为应如同 POSIX.1-2024 卷中定义的任何函数都不调用 `setlocale()`。

`setlocale()` 函数不需要是线程安全的；但是，它应避免与所有不影响且不受全局本地化环境影响的函数调用发生数据竞争。

返回值

成功完成时，`setlocale()` 应返回与新本地化环境的指定类别相关联的字符串。否则，`setlocale()` 应返回空指针且全局本地化环境不应更改。

`locale` 的空指针将导致 `setlocale()` 返回一个指向与当前全局本地化环境的指定 `category` 相关联的字符串的指针。全局本地化环境不应更改。

`setlocale()` 返回的字符串应使得使用该字符串及其关联的 `category` 的后续调用应恢复全局本地化环境的该部分。应用程序不应修改返回的字符串。返回的字符串指针可能会失效，或者字符串内容可能会被后续的 `setlocale()` 调用覆盖。如果调用线程终止，返回的指针也可能失效。

错误

未定义错误。

应用程序用法

以下代码说明程序如何为一种语言初始化国际化环境，同时选择性地修改全局本地化环境，使得正则表达式和字符串操作可以应用于以不同语言记录的文本：

```
setlocale(LC_ALL, "De");
setlocale(LC_COLLATE, "Fr@dict");
```

国际化程序可以通过按以下方式调用 `setlocale()` 来根据环境变量设置启动语言操作：

```
setlocale(LC_ALL, "");
```

更改 `LC_MESSAGES` 的设置对已通过 `catopen()` 调用打开的目录没有影响。

为了在多线程运行时使用不同的本地化环境设置，应用程序应优先使用 `uselocale()` 而不是 `setlocale()`。

原理

下文中对国际化环境或本地化环境的引用涉及进程的全局本地化环境。这可以使用 `uselocale()` 为单个线程覆盖。

ISO C 标准定义了一组支持国际化的函数。这些函数最重要的方面之一是设置和查询国际化环境的设施。国际化环境是影响某些功能行为的信息存储库，即：

1. 字符处理
2. 排序
3. 日期/时间格式化
4. 数值编辑

5. 货币格式化

6. 消息传递

`setlocale()` 函数为应用程序开发人员提供了设置国际化环境的全部或部分(称为类别)的能力。这些类别对应于上述功能区域。

`setlocale()` 有两个主要用途：

1. 查询国际化环境以了解其设置内容
2. 将国际化环境或本地化环境设置为特定值

要查询国际化环境，使用特定类别和空指针作为本地化环境来调用 `setlocale()`。空指针是 `setlocale()` 的特殊指令，告诉它查询而不是设置国际化环境。

有三种方式使用 `setlocale()` 设置国际化环境：

- `setlocale(category, string)`

此用法将国际化环境中的特定 `category` 设置为对应于 `string` 值的特定值。例如：

```
setlocale(LC_ALL, "fr_FR.ISO-8859-1");
```

- `setlocale(category, "C")`

ISO C 标准规定所有符合实现的实现上必须存在一个本地化环境。本地化环境的名称是 C，对应于支持 C 编程语言所需的最小国际化环境。

- `setlocale(category, "")`

这将特定类别设置为实现定义的默认值。这对应于环境变量的值。

参见

`catopen()`, `exec`, `fprintf()`, `fscanf()`, `getlocalename_l()`,
`isalnum()`, `isalpha()`, `isblank()`, `iscntrl()`, `isdigit()`,
`isgraph()`, `islower()`, `isprint()`, `ispunct()`, `isspace()`,
`isupper()`, `iswalnum()`, `iswalpha()`, `iswblank()`, `iswcntrl()`,
`iswctype()`, `iswdigit()`, `iswgraph()`, `iswlower()`, `iswprint()`,
`iswpunct()`, `iswspace()`, `iswupper()`, `iswdxdigit()`, `isxdigit()`,
`localeconv()`, `mblen()`, `mbstowcs()`, `mbtowc()`, `newlocale()`,
`nl_langinfo()`, `perror()`, `psiginfo()`, `strcoll()`, `strerror()`,
`strfmon()`, `strsignal()`, `strtod()`, `strxfrm()`, `tolower()`,

`toupper()` , `towlower()` , `towupper()` , `use locale()` , `wcs coll()` ,
`wcstod()` , `wcstombs()` , `wcsxfrm()` , `wctomb()`

XBD 7. 本地化环境, 8. 环境变量, `<langinfo.h>` , `<locale.h>`

更改历史

首次发布于第 3 期。在后续期中更新, 以与 POSIX 线程扩展对齐, 标记超出 ISO C 标准的扩展, 并应用各种技术更正。

1.205. setvbuf — 为流分配缓冲区

SYNOPSIS (函数概要)

```
#include <stdio.h>

int setvbuf(FILE *restrict stream, char *restrict buf, int type
```

DESCRIPTION (函数说明)

`setvbuf()` 函数可以在由 `stream` 指向的流与打开的文件关联之后，但在对该流执行任何其他操作（除了对 `setvbuf()` 的不成功调用之外）之前使用。

参数 `type` 确定 `stream` 应如何进行缓冲，如下所示：

- `_IOFBF` 将导致输入/输出完全缓冲。
- `_IOLBF` 将导致输入/输出行缓冲。
- `_IONBF` 将导致输入/输出无缓冲。

如果 `buf` 不是空指针，则它指向的数组可以用来代替由 `setvbuf()` 分配的缓冲区，参数 `size` 指定数组的大小；否则，`size` 可以确定由 `setvbuf()` 函数分配的缓冲区的大小。数组在任何时候的内容都是未指定的。

有关流的信息，请参见 [2.5 标准I/O流](#)。

RETURN VALUE (返回值)

成功完成后，`setvbuf()` 应返回 0。否则，如果为 `type` 给定了无效值或无法满足请求，它应返回非零值，并可能设置 `errno` 来指示错误。

ERRORS (错误条件)

`setvbuf()` 函数可能会失败：

- **EBADF**: 流不是内存流，且流底层的文件描述符无效。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

一个常见的错误源是在代码块中将缓冲区空间作为"自动"变量分配，然后未能在同一代码块中关闭流。

使用 `setvbuf()` 时，分配 `size` 字节的缓冲区并不一定意味着所有 `size` 字节都用于缓冲区。

应用程序应注意，许多实现对来自终端设备的输入只提供行缓冲。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- [2.5 标准I/O流](#)
- [fopen\(\)](#)
- [setbuf\(\)](#)
-

CHANGE HISTORY (变更历史)

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 6

标记了超越 ISO C 标准的扩展。

`setvbuf()` 原型已更新，以与 ISO/IEC 9899:1999 标准对齐。

Issue 8

应用了 Austin Group Defect 1144，更改了 [EBADF] 错误条件。

1.206. `shm_open` — 打开共享内存对象 (REALTIME)

概要

```
[SHM] #include <sys/mman.h>

int shm_open(const char *name, int oflag, mode_t mode);
```

描述

`shm_open()` 函数应建立共享内存对象与文件描述符之间的连接。它应创建一个引用共享内存对象的打开文件描述，以及一个引用该打开文件描述的文件描述符。文件描述符应按照 2.6 文件描述符分配 中的描述进行分配，并可被其他函数用来引用该共享内存对象。

`name` 参数指向一个命名共享内存对象的字符串。名称是否出现在文件系统中并对以路径名作为参数的其他函数可见是未指定的。`name` 参数符合路径名的构造规则，但 `name` 中除前导 `<slash>` 字符外的 `<slash>` 字符的解释是实现定义的，且 `name` 参数的长度限制是实现定义的，不需要与路径名限制 `{PATH_MAX}` 和 `{NAME_MAX}` 相同。

如果 `name` 以 `<slash>` 字符开头，那么调用 `shm_open()` 时使用相同 `name` 值的进程将引用同一个共享内存对象，只要该名称未被删除。如果 `name` 不以 `<slash>` 字符开头，则效果是实现定义的。

如果成功，`shm_open()` 应返回共享内存对象的文件描述符。打开文件描述是新的，因此文件描述符不与任何其他进程共享它。文件偏移量是否设置是未指定的。与新文件描述符关联的 `FD_CLOEXEC` 文件描述符标志应被设置。

打开文件描述的文件状态标志和文件访问模式应根据 `oflag` 的值进行设置。`oflag` 参数是以下标志的按位包含 OR。应用程序应在 `oflag` 的值中恰好指定以下前两个值（访问模式）之一：

访问模式

- `0_RDONLY` - 仅以读访问方式打开。
- `0_RDWR` - 以读或写访问方式打开。

附加标志

`oflag` 的值中可以指定剩余标志的任意组合：

- `O_CREAT` - 如果共享内存对象存在，此标志无效，除了如下面 `O_EXCL` 中所述的情况外。否则，创建共享内存对象。共享内存对象的用户 ID 应设置为进程的有效用户 ID。共享内存对象的组 ID 应设置为进程的有效组 ID；但是，如果 `name` 参数在文件系统中可见，组 ID 可以设置为包含目录的组 ID。共享内存对象的权限位应设置为 `mode` 参数的值，但进程的文件模式创建掩码中设置的位除外。当设置了除文件权限位外的 `mode` 中的位时，效果是未指定的。`mode` 参数不影响共享内存对象是为读取、为写入还是为两者打开。共享内存对象的大小为零。
- `O_EXCL` - 如果设置了 `O_EXCL` 和 `O_CREAT`，且共享内存对象存在，则 `shm_open()` 失败。对于其他执行 `shm_open()` 并使用相同共享内存对象名称且设置了 `O_EXCL` 和 `O_CREAT` 的进程，检查共享内存对象是否存在以及创建对象（如果不存在）的操作是原子的。如果设置了 `O_EXCL` 但未设置 `O_CREAT`，则结果是未定义的。
- `O_TRUNC` - 如果共享内存对象存在，并且它成功以 `O_RDWR` 方式打开，则对象应被截断为零长度，模式和工作所有者应不被此函数调用更改。将 `O_TRUNC` 与 `O_RDONLY` 一起使用的结果是未定义的。

原子操作

以下函数在对共享内存对象操作时，在 POSIX.1-2024 中指定的效果方面应彼此原子：

- `close()`
- `ftruncate()`
- `mmap()`
- `shm_open()`
- `shm_unlink()`

如果两个线程各自调用这些函数中的一个，每个调用要么看到另一个调用的所有指定效果，要么都不看到。对 `close()` 函数的要求也应适用于文件描述符成功关闭的任何情况，无论是什么原因（例如，作为调用 `close()`、调用 `dup2()` 或进程终止的结果）。

当创建共享内存对象时，共享内存对象的状态，包括与共享内存对象关联的所有数据，将持续存在，直到共享内存对象被取消链接且所有其他引用都消失。系统重启后名称和共享内存对象状态是否保持有效是未指定的。

返回值

成功完成后，`shm_open()` 函数应返回表示文件描述符的非负整数。否则，它应返回 -1 并设置 `errno` 以指示错误。

错误

如果出现以下情况，`shm_open()` 函数应失败：

- **[EACCES]** - 共享内存对象存在且 `oflag` 指定的权限被拒绝，或者共享内存对象不存在且创建共享内存对象的权限被拒绝，或者指定了 `O_TRUNC` 且写权限被拒绝。
- **[EEXIST]** - 设置了 `O_CREAT` 和 `O_EXCL` 且命名的共享内存对象已存在。
- **[EINTR]** - `shm_open()` 操作被信号中断。
- **[EINVAL]** - 指定名称不支持 `shm_open()` 操作。
- **[EMFILE]** - 进程可用的所有文件描述符当前都已打开。
- **[ENFILE]** - 系统中当前打开的共享内存对象过多。
- **[ENOENT]** - 未设置 `O_CREAT` 且命名的共享内存对象不存在。
- **[ENOSPC]** - 创建新共享内存对象的空间不足。

如果出现以下情况，`shm_open()` 函数可能失败：

- **[ENAMETOOLONG]** - `name` 参数的长度在不支持 XSI 选项 [XSI] 的系统上超过 `{_POSIX_PATH_MAX}` 或在 XSI 系统上超过 `{_XOPEN_PATH_MAX}`，或者具有在不支持 XSI 选项 [XSI] 的系统上长于 `{_POSIX_NAME_MAX}` 或在 XSI 系统上长于 `{_XOPEN_NAME_MAX}` 的路径名组件。

示例

创建和映射共享内存对象

以下代码段演示了使用 `shm_open()` 创建共享内存对象，然后使用 `ftruncate()` 设置其大小，最后使用 `mmap()` 将其映射到进程地址空间：

```
#include <unistd.h>
#include <sys/mman.h>
...
```

```
#define MAX_LEN 10000
struct region {           /* 定义共享内存的"结构" */
    int len;
    char buf[MAX_LEN];
};
struct region *rptr;
int fd;

/* 创建共享内存对象并设置其大小 */

fd = shm_open("/myregion", O_CREAT | O_RDWR, S_IRUSR | S_IWUSR)
if (fd == -1)
    /* 处理错误 */;

if (ftruncate(fd, sizeof(struct region)) == -1)
    /* 处理错误 */;

/* 映射共享内存对象 */

rptr = mmap(NULL, sizeof(struct region),
            PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
if (rptr == MAP_FAILED)
    /* 处理错误 */;

/* 现在我们可以使用 rptr 的字段引用映射区域;
   例如, rptr->len */

...
```

应用程序用法

无。

基本原理

当支持内存映射文件选项时，使用正常的 `open()` 调用来获取要映射文件的描述符，这是与 `mmap()` 的现有实践保持一致。当支持共享内存对象选项时，`shm_open()` 函数应获取要映射的共享内存对象的描述符。

用文件描述符表示几种类型的对象有充分的先例。在 POSIX.1-1990 标准中，文件描述符可以表示文件、管道、FIFO、tty 或目录。许多实现简单地有一个操作向量，该向量按文件描述符类型索引并执行非常不同的操作。请注意，在某些情况下，传递给文件描述符泛型操作的文件描述符由 `open()` 或 `creat()` 返回。

回，在某些情况下由替代函数（如 `pipe()`）返回。`shm_open()` 使用后一种技术。

请注意，此类共享内存对象实际上可以实现为映射文件。在这两种情况下，都可以在打开后使用 `ftruncate()` 设置大小。`shm_open()` 函数本身不创建指定大小的共享对象，因为这将重复设置文件描述符引用对象大小的现有函数。

在内存对象使用现有文件系统实现的实现上，`shm_open()` 函数可以使用调用 `open()` 的宏来实现，而 `shm_unlink()` 函数可以使用调用 `unlink()` 的宏来实现。

对于没有永久文件系统的实现，允许内存对象的名称定义在系统重启后不保留。请注意，这允许具有永久文件系统的系统也将内存对象实现为实现内部的数据结构。

在选择直接使用内存实现内存对象的实现上，`shm_open()` 后跟 `ftruncate()` 和 `close()` 可用于预分配共享内存区域并设置该预分配的大小。这对于没有虚拟内存硬件支持的系统可能是必要的，以确保内存是连续的。

`shm_open()` 的有效打开标志集被限制为 `O_RDONLY`、`O_RDWR`、`O_CREAT` 和 `O_TRUNC`，因为这些可以在大多数内存映射系统上轻松实现。本 POSIX.1-2024 卷对实现因实现定义的原因（包括硬件原因）无法提供请求的文件访问时的结果保持沉默。`O_CLOEXEC` 打开标志未列出，因为所有共享内存对象创建时都已设置 `FD_CLOEXEC` 标志；应用程序稍后可以使用 `fcntl()` 清除该标志，以允许共享内存文件描述符在 `exec` 函数族中保留。

错误条件 `[EACCES]` 和 `[ENOTSUP]` 的提供是为了告知应用程序实现无法完成请求。

`[EACCES]` 指示由于实现定义的原因（可能是硬件相关），实现无法遵守请求的模式，因为它与另一个请求的模式冲突。一个例子可能是应用程序希望打开内存对象两次，使用不同的访问模式映射不同的区域。如果实现无法将单个区域映射到进程空间中的两个位置（如果两个区域需要不同的访问模式，这是必需的），那么实现可以在第二次打开时通知应用程序。

`[ENOTSUP]` 指示由于实现定义的原因（可能是硬件相关），实现完全无法遵守请求的模式。一个例子是实现的硬件不支持只写共享内存区域。

在所有实现上，为了性能（如 RAM 磁盘）或实现定义的原因（仅在特定文件系统上支持共享内存），可能需要将内存对象的位置限制在特定的文件系统。

`shm_open()` 函数可用于强制执行这些限制。应用程序有多种方法可用于确定文件的适当名称或适当目录的位置。一种方法是通过环境变量使用 `getenv()`。另一种方法是来自配置文件。

本 POSIX.1-2024 卷指定内存对象在创建时初始内容为零。这与当前文件和新分配内存的行为一致。对于那些使用物理内存的实现，此类实现可能只是简单地使用可用内存并将其未初始化地给予进程。但是，这与未初始化数据区域、堆栈以及文件的标准行为不一致。最后，出于安全原因，将分配的内存设置为零是非常可取的。因此，需要将内存对象初始化为零。

未来方向

未来版本可能要求 `shm_open()` 和 `shm_unlink()` 函数具有类似于正常文件系统操作的语义。

另请参见

- 2.6 文件描述符分配
- `close()`
- `dup()`
- `exec`
- `fcntl()`
- `mmap()`
- `shmat()`
- `shmctl()`
- `shmdt()`
- `shm_unlink()`
- `umask()`
- XBD `<fcntl.h>`
- XBD `<sys/mman.h>`

变更历史

首次在 Issue 5 中发布。包含用于与 POSIX 实时扩展对齐。

Issue 6

`shm_open()` 函数被标记为共享内存对象选项的一部分。

[ENOSYS] 错误条件已被删除，因为如果实现不支持共享内存对象选项，则不需要提供存根。

IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/126 被应用，在示例部分添加了示例。

Issue 7

Austin Group 解释 1003.1-2001 #077 被应用，阐明了 **name** 参数并将 **[ENAMETOOLONG]** 从"应失败"错误更改为"可能失败"错误。

Austin Group 解释 1003.1-2001 #141 被应用，添加了未来方向。

SD5-XSH-ERN-170 被应用，更新描述以阐明设置共享内存对象用户 ID 和组 ID 的措辞。

POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0324 [835]、XSH/TC2-2008/0325 [835] 和 XSH/TC2-2008/0326 [835] 被应用。

Issue 8

Austin Group 缺陷 411 被应用，在基本原理部分添加了关于 **0_CLOEXEC** 的句子。

Austin Group 缺陷 593 被应用，从描述部分删除了对 **<fcntl.h>** 的引用。

Austin Group 缺陷 695 被应用，为共享内存对象上的操作添加了原子性要求。

1.207. `shm_unlink` — 删 除 共 享 内 存 对 象 (REALTIME)

概要 (SYNOPSIS)

```
#include <sys/mman.h>

int shm_unlink(const char *name);
```

描述 (DESCRIPTION)

`shm_unlink()` 函数应删除由 `name` 指针指向的字符串命名的共享内存对象的名称。

如果在解除链接 (unlink) 时共享内存对象存在一个或多个引用，则名称应在 `shm_unlink()` 返回之前被删除，但内存对象内容的删除应推迟到所有对该共享内存对象的打开和映射引用都被移除之后。

即使对象在最后一次 `shm_unlink()` 之后继续存在，后续重用该名称应导致 `shm_open()` 表现得好像不存在此名称的共享内存对象（即，如果未设置 `O_CREAT`，`shm_open()` 应失败，或者如果设置了 `O_CREAT`，应创建一个新的共享内存对象）。

返回值 (RETURN VALUE)

成功完成时，应返回零值。否则，应返回 -1 并设置 `errno` 以指示错误。如果返回 -1，则命名的共享内存对象不应被此函数调用修改。

错误 (ERRORS)

`shm_unlink()` 函数在以下情况下应失败：

- **[EACCES]** - 拒绝解除链接命名共享内存对象的权限。
- **[ENOENT]** - 命名的共享内存对象不存在。

`shm_unlink()` 函数在以下情况下可能失败：

- **[ENAMETOOLONG]** - `name` 参数的长度在不支持 XSI 选项的系统上超过 `{_POSIX_PATH_MAX}` 或在 XSI 系统上超过 `{_XOPEN_PATH_MAX}`，或者

包含的路径名组件在不支持 XSI 选项的系统上长于 `{_POSIX_NAME_MAX}` 或在 XSI 系统上长于 `{_XOPEN_NAME_MAX}`。使用包含与之前在成功的 `shm_open()` 调用中使用的相同共享内存对象名称的 `name` 参数调用 `shm_unlink()` 不应给出 [ENAMETOOLONG] 错误。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

使用 `open()` 分配的内存对象名称以通常方式用 `unlink()` 删除。使用 `shm_open()` 分配的内存对象名称用 `shm_unlink()` 删除。请注意，如果内存对象已经在使用中，实际的内存对象在最后一次关闭和取消映射之前不会被销毁。

基本原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

未来版本可能要求 `shm_open()` 和 `shm_unlink()` 函数具有与普通文件系统操作类似的语义。

另请参阅 (SEE ALSO)

- `close()`
- `mmap()`
- `munmap()`
- `shmat()`
- `shmctl()`
- `shmdt()`
- `shm_open()`

- `<sys/mman.h>`

变更历史 (CHANGE HISTORY)

第 5 版首次发布

为与 POSIX 实时扩展对齐而包含。

第 6 版

`shm_unlink()` 函数被标记为共享内存对象选项的一部分。

在描述 (DESCRIPTION) 中, 添加了文本以阐明在 `shm_unlink()` 后重用相同名称不会附加到旧的共享内存对象。

[ENOSYS] 错误条件已被移除, 因为如果实现不支持共享内存对象选项, 则不需要提供存根 (stubs)。

第 7 版

应用 Austin Group 解释 1003.1-2001 #077, 将 [ENAMETOOLONG] 从 "应失败" 错误更改为 "可能失败" 错误。

应用 Austin Group 解释 1003.1-2001 #141, 添加未来方向 (FUTURE DIRECTIONS)。

1.208. `sigaction`

概要

```
#include <signal.h>

int sigaction(int sig, const struct sigaction *restrict act,
              struct sigaction *restrict oact);
```

描述

`sigaction()` 函数允许调用进程检查和/或指定与特定信号相关联的动作。参数 `sig` 指定信号；可接受的值在 `<signal.h>` 中定义。

用于描述要执行动作的 `sigaction` 结构在 `<signal.h>` 头文件中定义，至少包含以下成员：

成员类型	成员名称	描述
<code>void (*)(int)</code>	<code>sa_handler</code>	指向信号捕获函数的指针，或宏 <code>SIG_IGN</code> 或 <code>SIG_DFL</code> 之一。
<code>sigset_t</code>	<code>sa_mask</code>	在执行信号捕获函数期间要阻塞的附加信号集合。
<code>int</code>	<code>sa_flags</code>	影响信号行为的特殊标志。
<code>void (*)(int, siginfo_t *, void *)</code>	<code>sa_sigaction</code>	指向信号捕获函数的指针。

`sa_handler` 和 `sa_sigaction` 占用的存储空间可能重叠，符合规范的应用程序不应同时使用两者。

如果参数 `act` 不是空指针，它指向一个结构，该结构指定要与指定信号相关联的动作。如果参数 `oact` 不是空指针，则先前与该信号相关联的动作将存储在参数 `oact` 指向的位置。如果参数 `act` 是空指针，则信号处理保持不变；因此，该调用可用于查询给定信号的当前处理方式。SIGKILL 和 SIGSTOP 信号不应使用此机制添加到信号掩码中；此限制应由系统强制执行，而不会指示错误。

如果在 `sigaction` 结构的 `sa_flags` 字段中清除了 `SA_SIGINFO` 标志（见下文），则 `sa_handler` 字段标识要与指定信号相关联的动作。如果在 `sa_flags` 字段中设置了 `SA_SIGINFO` 标志，则 `sa_sigaction` 字段指定信号捕获函数。

`sa_flags` 字段可用于修改指定信号的行为。

以下在 `<signal.h>` 头文件中定义的标志可以在 `sa_flags` 中设置：

SA_NOCLDSTOP

当子进程停止[XSI]或已停止的子进程继续时，不产生 `SIGCHLD`。

如果 `sig` 是 `SIGCHLD` 且未在 `sa_flags` 中设置 `SA_NOCLDSTOP` 标志，并且实现支持 `SIGCHLD` 信号，那么当其任何子进程停止[XSI]时，应为调用进程生成 `SIGCHLD` 信号，并且当其任何已停止的子进程继续时，可以为调用进程生成 `SIGCHLD` 信号。如果 `sig` 是 `SIGCHLD` 且在 `sa_flags` 中设置了 `SA_NOCLDSTOP` 标志，则实现不应以这种方式生成 `SIGCHLD` 信号。

SA_ONSTACK

[XSI] 如果设置且已使用 `sigaltstack()` 声明了替代信号栈，则信号应在该栈上传递给调用进程。否则，信号应在当前栈上传递。

SA_RESETHAND

如果设置，信号的处置应在进入信号处理程序时重置为 `SIG_DFL`，并清除 `SA_SIGINFO` 标志。

注意： `SIGILL` 和 `SIGTRAP` 在传递时不能自动重置；系统会静默地强制执行此限制。

否则，信号的处置在进入信号处理程序时不应被修改。

此外，如果设置了此标志，`sigaction()` 的行为可能如同也设置了 `SA_NODEFER` 标志。

SA_RESTART

此标志影响可中断函数的行为；即那些被指定为因 `errno` 设置为 `[EINTR]` 而失败的函数。如果设置，且被指定为可中断的函数被此信号中断，则函数应重新启动且不会因 `[EINTR]` 而失败，除非另有指定。如果使用超时的可中断函数重新启动，则重新启动后的超时持续时间被设置为不超过原始超时值的未指定

值。如果未设置标志，被此信号中断的可中断函数应因 `errno` 设置为 [EINTR] 而失败。

SA_SIGINFO

如果清除且捕获了信号，信号捕获函数应按以下方式进入：

```
void func(int signo);
```

其中 `signo` 是信号捕获函数的唯一参数。在这种情况下，应用程序应使用 `sa_handler` 成员来描述信号捕获函数，且应用程序不应修改 `sa_sigaction` 成员。

如果设置了 SA_SIGINFO 且捕获了信号，信号捕获函数应按以下方式进入：

```
void func(int signo, siginfo_t *info, void *context);
```

其中两个附加参数被传递给信号捕获函数。第二个参数应指向 `siginfo_t` 类型的对象，解释生成信号的原因；第三个参数可以转换为指向 `ucontext_t` 类型对象的指针，以引用在信号传递时被中断的接收线程的上下文。在这种情况下，应用程序应使用 `sa_sigaction` 成员来描述信号捕获函数，且应用程序不应修改 `sa_handler` 成员。

`si_signo` 成员包含系统生成的信号编号。

[XSI] `si_errno` 成员可能包含实现定义的附加错误信息；如果非零，它包含一个错误编号，标识导致信号生成的条件。

`si_code` 成员包含标识信号原因的代码，如 2.4.3 信号动作 中所述。

SA_NOCLDWAIT

[XSI] 如果 `sig` 不等于 SIGCHLD，则行为未指定。否则，SA_NOCLDWAIT 标志的行为如 进程终止的后果 中所述。

SA_NODEFER

如果设置且捕获了 `sig`，则 `sig` 不应在线程进入信号处理程序时添加到线程的信号掩码中，除非它包含在 `sa_mask` 中。否则，`sig` 应始终在进入信号处理程序时添加到线程的信号掩码中。

当信号被 `sigaction()` 安装的信号捕获函数捕获时，会计算新的信号掩码并在信号捕获函数期间（或直到调用 `sigprocmask()` 或 `sigsuspend()`）安装。此掩码通过取当前信号掩码与正在传递的信号的 `sa_mask` 值的并集形

成，并且除非设置了 SA_NODEFER 或 SA_RESETHAND，还包括正在传递的信号。当用户的信号处理程序正常返回时，原始信号掩码被恢复。

一旦为特定信号安装了动作，它应保持安装状态，直到明确请求另一个动作（通过另一次 `sigaction()` 调用），直到 SA_RESETHAND 标志导致处理程序重置，或者直到调用某个 `exec` 函数。

如果 `sig` 的先前动作是由 `signal()` 建立的，则在 `oact` 指向的结构中返回的字段值是未指定的，特别是 `oact->sa_handler` 不一定是传递给 `signal()` 的相同值。但是，如果指向相同结构或其副本的指针通过 `act` 参数传递给后续的 `sigaction()` 调用，则信号的处理应如同重复对 `signal()` 的原始调用。

如果 `sigaction()` 失败，则不会安装新的信号处理程序。

将无法捕获或忽略的信号的动作设置为 SIG_DFL 的尝试是被忽略还是导致返回错误且 `errno` 设置为 [EINVAL] 是未指定的。

如果未在 `sa_flags` 中设置 SA_SIGINFO，则当 `sig` 已处于挂起状态时后续出现的信号的处置是实现定义的；信号捕获函数应使用单个参数调用。如果在 `sa_flags` 中设置了 SA_SIGINFO，则由 `sigqueue()` 生成的或作为支持指定应用程序定义值的任何信号生成函数结果的后续 `sig` 出现（当 `sig` 已处于挂起状态时）应按 FIFO 顺序排队，直到传递或接受；信号捕获函数应使用三个参数调用。应用程序指定的值作为 `siginfo_t` 结构的 `si_value` 成员传递给信号捕获函数。

在同一进程中同时使用 `sigaction()` 和 `sigwait()` 函数处理相同信号的结果是未指定的。

返回值

成功完成后，`sigaction()` 应返回 0；否则应返回 -1，`errno` 应设置为指示错误，且不会安装新的信号捕获函数。

错误

`sigaction()` 函数在以下情况下应失败：

- [EINVAL] - `sig` 参数不是有效的信号编号，或者尝试捕获无法捕获的信号或忽略无法忽略的信号。

`sigaction()` 函数在以下情况下可能失败：

- **[EINVAL]** - 尝试将无法捕获或忽略（或两者）的信号的动作设置为 SIG_DFL。

此外，在支持 XSI 选项的系统上，如果对于不在 SIGRTMIN 到 SIGRTMAX 范围内的信号，在 **sigaction** 结构的 **sa_flags** 字段中设置了 SA_SIGINFO 标志，**sigaction()** 函数可能失败。

以下章节为信息性内容。

示例

建立信号处理程序

以下示例演示了使用 **sigaction()** 为 SIGINT 信号建立处理程序。

```
#include <signal.h>

static void handler(int signum)
{
    /* 对信号传递采取适当的行动 */
}

int main(void)
{
    struct sigaction sa;

    sa.sa_handler = handler;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = SA_RESTART; /* 如果被处理程序中断，重新启动函数 */
    if (sigaction(SIGINT, &sa, NULL) == -1)
        /* 处理错误 */;

    /* 进一步的代码 */
}
```

应用程序使用

sigaction() 函数取代了 **signal()** 函数，应优先使用。特别是，**sigaction()** 和 **signal()** 不应在同一进程中用于控制相同信号。异步信号安全函数的行为，如在其各自的 DESCRIPTION 部分中所定义的，如 POSIX.1-2024 本卷所指定的，无论是否从信号捕获函数调用。这就是异步信号安全函数可以在信号捕获函数中无限制使用的语句的唯一预期含义。应用程序仍必须考

虑此类函数对数据结构、文件和进程状态等所有影响。特别是，应用程序开发人员需要考虑中断 `sleep()` 时的交互限制以及文件描述符的多个句柄之间的交互。任何特定函数被列为异步信号安全的事实并不一定意味着推荐从信号捕获函数调用该函数。

为防止中断非异步信号安全函数调用而产生的错误，应用程序应通过阻塞适当的信号或通过使用某些程序信号量（参见 `semget()`、`sem_init()`、`sem_open()` 等）来保护对这些函数的调用。特别要注意，即使是“安全”函数也可能修改 `errno`；信号捕获函数，如果不作为独立线程执行，应保存和恢复其值，以避免在设置 `errno` 的函数错误返回与随后的 `errno` 检查之间传递信号可能导致信号捕获函数更改 `errno` 值的可能性。自然，相同的原理适用于应用程序例程和异步数据访问的异步信号安全性。注意 `longjmp()` 和 `siglongjmp()` 不在异步信号安全函数列表中。这是因为在 `longjmp()` 和 `siglongjmp()` 之后执行的代码可以调用任何不安全的函数，其危险性与直接从信号处理程序调用这些不安全函数相同。在信号处理程序中使用 `longjmp()` 和 `siglongjmp()` 的应用程序需要严格保护才能具有可移植性。许多被排除在列表之外的其他函数传统上使用 `malloc()` 或 `free()` 函数或标准 I/O 库实现，这两者传统上以非异步信号安全的方式使用数据结构。由于使用公共数据结构的不同函数的任何组合都可能导致异步信号安全问题，POSIX.1-2024 本卷未定义在中断不安全函数的信号处理程序中调用任何不安全函数时的行为。

通常，信号在信号传递之前生效的栈上执行。可以指定替代栈来接收被捕获信号的子集。

当信号处理程序返回时，接收线程在被中断的点恢复执行，除非信号处理程序做出其他安排。如果使用 `longjmp()` 离开信号处理程序，则必须显式恢复信号掩码。

POSIX.1-2024 本卷将设置 `SA_SIGINFO` 时信号处理函数的第三个参数定义为 `void *` 而不是 `ucontext_t *`，但不要求类型检查。新应用程序应将信号处理函数的第三个参数显式转换为 `ucontext_t *`。

本卷 POSIX.1-2024 不支持 BSD 可选的四参数信号处理函数。BSD 声明应为：

```
void handler(int sig, int code, struct sigcontext *scp, char *a
```

其中 `sig` 是信号编号，`code` 是某些信号的附加信息，`scp` 是指向 `sigcontext` 结构的指针，`addr` 是附加地址信息。在设置 `SA_SIGINFO` 时指定的信号处理程序的第二个参数指向的对象中提供了大致相同的信息。

由于 `sigaction()` 函数在应用程序设置 `SA_RESETHAND` 标志时允许但不要求设置 `SA_NODEFER`，因此依赖新安装信号处理程序的 `SA_RESETHAND` 功能

的应用程序必须在设置 SA_RESETHAND 时始终显式设置 SA_NODEFER 才能具有可移植性。

另请参见 XRAT B.2.4.2 实时信号生成和传递中关于实时信号生成和传递的基本原理。

基本原理

尽管 POSIX.1-2024 本卷要求不可忽略的信号在进入信号捕获函数时不应添加到信号掩码中，但没有明确要求后续的 `sigaction()` 调用在 `oact` 参数返回的信息中反映这一点。换句话说，如果 SIGKILL 包含在 `act` 的 `sa_mask` 字段中，则后续的 `sigaction()` 调用返回时 `oact` 的 `sa_mask` 字段中是否包含 SIGKILL 是未指定的。

在 `act->sa_flags` 参数中提供 SA_NOCLDSTOP 标志时，允许用 System V 语义重载 SIGCHLD，即每个 SIGCLD 信号指示一个终止的子进程。大多数捕获 SIGCHLD 的符合规范的应用程序预期安装信号捕获函数，该函数重复调用设置了 WNOHANG 标志的 `waitpid()` 函数，对返回状态的每个子进程采取行动，直到 `waitpid()` 返回零。如果对已停止的子进程不感兴趣，使用 SA_NOCLDSTOP 标志可以防止在它们停止时调用信号捕获例程的开销。

一些历史实现还定义了停止进程的其他机制，如 `ptrace()` 函数。这些实现通常不...

1.209. sigaddset

SYNOPSIS

```
#include <signal.h>

int sigaddset(sigset_t *set, int signo);
```

DESCRIPTION

`sigaddset()` 函数将由 `signo` 指定的单个信号添加到 `set` 指向的信号集中。

应用程序在首次使用任何 `sigset_t` 类型的对象之前，必须至少调用一次 `sigemptyset()` 或 `sigfillset()` 函数来初始化该对象。如果这样的对象未通过此方式初始化，但仍作为参数提供给 `pthread_sigmask()`、
`sigaction()`、`sigaddset()`、`sigdelset()`、`sigismember()`、
`sigpending()`、`sigprocmask()`、`sigsuspend()`、
`sigtimedwait()`、`sigwait()` 或 `sigwaitinfo()` 中的任何一个函数，则结果是未定义的。

RETURN VALUE

成功完成后，`sigaddset()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

ERRORS

`sigaddset()` 函数可能在以下情况下失败：

- **[EINVAL]**

`signo` 参数的值是无效或不支持的信号编号。

以下章节是参考性内容。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [2.4 Signal Concepts](#)
- [pthread_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigdelset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)
- [sigismember\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
- XBD

CHANGE HISTORY

首次发布于 Issue 3。

为与 POSIX.1-1988 标准对齐而包含。

Issue 5

DESCRIPTION 的最后一段在之前的版本中作为 APPLICATION USAGE 注释包含。

Issue 6

更新规范性文本以避免对应用程序要求使用术语 "must"。

SYNOPSIS 被标记为 CX，因为此函数在 头文件中的存在是对 ISO C 标准的扩展。

1.210. sigdelset - 从信号集中删除一个信号

SYNOPSIS (概要)

```
#include <signal.h>

int sigdelset(sigset_t *set, int signo);
```

DESCRIPTION (描述)

`sigdelset()` 函数从 `set` 指向的信号集中删除由 `signo` 指定的单个信号。

应用程序应在对任何 `sigset_t` 类型的对象进行任何其他使用之前，至少调用一次 `sigemptyset()` 或 `sigfillset()` 来初始化该对象。如果这样的对象没有以此方式初始化，但仍作为参数提供给 `pthread_sigmask()`、`sigaction()`、`sigaddset()`、`sigdelset()`、`sigismember()`、`sigpending()`、`sigprocmask()`、`sigsuspend()`、`sigtimedwait()`、`sigwait()` 或 `sigwaitinfo()` 中的任何一个函数，则结果未定义。

RETURN VALUE (返回值)

成功完成时，`sigdelset()` 应返回 0；否则，应返回 -1 并设置 `errno` 来指示错误。

ERRORS (错误)

`sigdelset()` 函数可能在以下情况下失败：

[EINVAL]

`signo` 参数不是有效的信号编号，或者是不支持的信号编号。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

无。

RATIONALE (原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- [Signal Concepts \(信号概念\)](#)
- [pthread_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)
- [sigismember\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
-

CHANGE HISTORY (变更历史)

首次发布于 Issue 3。为与 POSIX.1-1988 标准对齐而包含。

Issue 5

在之前的版本中，DESCRIPTION 的最后一段作为 APPLICATION USAGE 注释包含在内。

Issue 6

SYNOPSIS 被标记为 CX，因为此函数在 `<signal.h>` 头文件中的存在是对 ISO C 标准的扩展。

1.211. sigemptyset

SYNOPSIS

```
[CX] #include <signal.h>
int sigemptyset(sigset_t *set);
```

DESCRIPTION

`sigemptyset()` 函数初始化 `set` 所指向的信号集，使得所有在 POSIX.1-2024 中定义的信号都被排除。

RETURN VALUE

成功完成后，`sigemptyset()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

ERRORS

未定义任何错误。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

`sigemptyset()` (或 `sigfillset()`) 函数的实现可以很直接地清空 (或设置) 信号集中的所有位。另外, 初始化结构的部分内容 (如版本字段) 也是合理的, 这样可以允许在信号集大小不同的版本之间保持二进制兼容性。由于这些原因, 在信号集的任何其他使用之前, 必须调用 `sigemptyset()` 或 `sigfillset()`, 即使这种使用是只读的 (例如, 作为 `sigpending()` 的参数)。此函数不适用于动态分配。

`sigfillset()` 和 `sigemptyset()` 函数要求生成的信号集包含 (或排除) POSIX.1-2024 本卷中定义的所有信号。虽然 POSIX.1-2024 本卷的范围不包括对作为扩展实现的信号提出此要求, 但建议实现定义的信号也受这些函数的影响。然而, 可能有正当理由使某个特定信号不受影响。例如, 阻塞或忽略一个实现定义的信号可能会产生不良副作用, 而该信号的默认操作是无害的。在这种情况下, 最好将此类信号从 `sigfillset()` 返回的信号集中排除。

在早期的提案中, 无效信号和不支持信号之间没有区别 (实现不支持的可选信号名称不会被该实现定义)。因此, [EINVAL] 错误被指定为无效信号的必需错误。有了这种区别, 就不需要要求这些函数的实现来确定可选信号是否确实受支持, 因为这对性能可能有显著影响而价值很小。错误本可以要求对无效信号是必需的, 对不支持信号是可选的, 但这似乎过于复杂。因此, 在两种情况下错误都是可选的。

FUTURE DIRECTIONS

无。

SEE ALSO

2.4 信号概念, `pthread_sigmask()`, `sigaction()`, `sigaddset()`,
`sigdelset()`, `sigfillset()`, `sigismember()`, `sigpending()`,
`sigsuspend()`

XBD `<signal.h>`

CHANGE HISTORY

首次在 Issue 3 中发布。为了与 POSIX.1-1988 标准对齐而包含。

Issue 6

SYNOPSIS 被标记为 CX，因为此函数在 `<signal.h>` 头文件中的存在是对 ISO C 标准的扩展。

1.212. sigfillset - 初始化并填充信号集

SYNOPSIS

```
#include <signal.h>

int sigfillset(sigset_t *set);
```

DESCRIPTION

`sigfillset()` 函数应初始化 `set` 所指向的信号集，使其包含 POSIX.1-2024 本卷中定义的所有信号。

RETURN VALUE

成功完成时，`sigfillset()` 应返回 0；否则，应返回 -1 并设置 `errno` 来指示错误。

ERRORS

未定义错误。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

请参考 `sigemptyset()`。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.4 信号概念
- `pthread_sigmask()`
- `sigaction()`
- `sigaddset()`
- `sigdelset()`
- `sigemptyset()`
- `sigismember()`
- `sigpending()`
- `sigsuspend()`
- XBD `<signal.h>`

CHANGE HISTORY

首次在 Issue 3 中发布。为与 POSIX.1-1988 标准对齐而包含。

Issue 6

SYNOPSIS 被标记为 CX，因为此函数在 `<signal.h>` 头文件中的存在是对 ISO C 标准的扩展。

1.213. `sigismember`

SYNOPSIS

```
#include <signal.h>

int sigismember(const sigset_t *set, int signo);
```

DESCRIPTION

`sigismember()` 函数应测试由 `signo` 指定的信号是否是由 `set` 指向的集合的成员。

应用程序应在任何其他用途之前，对每个 `sigset_t` 类型的对象至少调用一次 `sigemptyset()` 或 `sigfillset()`。如果这样的对象没有通过这种方式初始化，但仍被作为参数提供给 `pthread_sigmask()`、`sigaction()`、`sigaddset()`、`sigdelset()`、`sigismember()`、`sigpending()`、`sigprocmask()`、`sigsuspend()`、`sigtimedwait()`、`sigwait()` 或 `sigwaitinfo()` 中的任何函数，则结果未定义。

RETURN VALUE

成功完成后，如果指定的信号是指定集合的成员，`sigismember()` 应返回 1，否则返回 0。否则，它应返回 -1 并设置 `errno` 来指示错误。

ERRORS

`sigismember()` 函数可能会失败，如果：

- **[EINVAL]**

`signo` 参数不是有效的信号编号，或者是不支持的信号编号。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [2.4 信号概念](#)
- [pthread_sigmask\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigdelset\(\)](#)
- [sigfillset\(\)](#)
- [sigemptyset\(\)](#)
- [sigpending\(\)](#)
- [sigsuspend\(\)](#)
-

CHANGE HISTORY

首次发布于 Issue 3

包含此项以与 POSIX.1-1988 标准保持一致。

Issue 5

DESCRIPTION 的最后一段在之前的版本中作为 APPLICATION USAGE 说明包含在内。

Issue 6

SYNOPSIS 被标记为 CX，因为此函数在 `<signal.h>` 头文件中的存在是对 ISO C 标准的扩展。

1.214. signal — 信号管理

概要

```
#include <signal.h>

void (*signal(int sig, void (*func)(int)))(int);
```

描述

`signal()` 函数选择三种方式之一来处理后续接收到的信号编号 `sig`。如果 `func` 的值为 `SIG_DFL`，则对该信号进行默认处理。如果 `func` 的值为 `SIG_IGN`，则该信号将被忽略。否则，应用程序应确保 `func` 指向一个当该信号发生时将被调用的函数。由于信号而调用这样的函数，或者（递归地）调用该调用所调用的任何其他函数（标准库中的函数除外），被称为"信号处理器"。

当信号发生且 `func` 指向一个函数时，是执行等价于：

```
signal(sig, SIG_DFL);
```

的语句，还是实现阻止某些实现定义的信号集合（至少包括 `sig`）在当前信号处理完成之前发生，这是由实现定义的。（如果 `sig` 的值为 `SIGILL`，实现也可以选择定义不采取任何操作。）接着执行等价于：

```
(*func)(sig);
```

的语句。如果函数返回时，`sig` 的值为 `SIGFPE`、`SIGILL` 或 `SIGSEGV` 或任何其他对应于计算异常的实现定义值，则行为是未定义的。否则，程序应在被中断的点恢复执行。

如果进程是多线程的，或者进程是单线程的且信号处理器不是作为以下操作的结果执行：

- 进程调用 `abort()`、`raise()`、`kill()`、`pthread_kill()` 或 `sigqueue()` 生成一个未阻塞的信号
- 一个待处理的信号被解除阻塞，并在解除阻塞的调用返回之前被传递

则在以下情况下行为是未定义的：

[其他行为条件在完整规范中继续...]

在程序启动时，对某些信号执行等价于：

```
signal(sig, SIG_IGN);
```

的语句，对所有其他信号执行等价于：

```
signal(sig, SIG_DFL);
```

的语句（参见 `exec`）。

如果 `signal()` 函数成功，不应改变 `errno` 的设置。

要求 `signal()` 函数是线程安全的。

返回值

如果请求可以被满足，`signal()` 应返回指定信号 `sig` 的最近一次 `signal()` 调用的 `func` 值。否则，应返回 `SIG_ERR` 并在 `errno` 中存储一个正值。

错误

如果出现以下情况，`signal()` 函数应失败：

- **[EINVAL]** `sig` 参数不是有效的信号编号，或者试图捕获无法捕获的信号，或忽略无法忽略的信号。

如果出现以下情况，`signal()` 函数可能失败：

- **[EINVAL]** 对于无法捕获或忽略（或两者都不可）的信号，试图将其操作设置为 `SIG_DFL`。

应用程序使用

`sigaction()` 函数提供了更全面和可靠的信号控制机制；新应用程序应使用 `sigaction()` 而不是 `signal()`。

原理

ISO C 标准规定，在多线程程序中使用 `signal()` 会导致未定义的行为。然而，在线程被添加到 ISO C 标准之前，POSIX.1 就已经要求 `signal()` 是线程

安全的。

参见

- [2.4 信号概念](#)
- [exec](#)
- [pause\(\)](#)
- [raise\(\)](#)
- [sigaction\(\)](#)
- [sigsuspend\(\)](#)
- [waitid\(\)](#)
-

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

从 X/OPEN UNIX 扩展移至 BASE。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0580 [275], XSH/TC1-2008/0581 [66], 和 XSH/TC1-2008/0582 [105]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0331 [785]。

Issue 8

应用了 Austin Group Defect 728，减少了信号处理器引用具有静态或线程存储持续时间的对象时导致未定义行为的情况集合。

应用了 Austin Group Defect 1302，使该函数与 ISO/IEC 9899:2018 标准保持一致。

1.215. `sigpending`

SYNOPSIS

```
#include <signal.h>

int sigpending(sigset_t *set);
```

DESCRIPTION

`sigpending()` 函数应将那些被阻塞无法传递给调用线程且在进程或调用线程上处于待处理状态的信号集合存储在 `set` 参数所引用的位置中。

RETURN VALUE

成功完成后, `sigpending()` 应返回 0; 否则, 应返回 -1 并设置 `errno` 以指示错误。

ERRORS

未定义错误。

以下部分为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

[exec](#),
[pthread_sigmask\(\)](#),
[sigaddset\(\)](#),
[sigdelset\(\)](#),
[sigemptyset\(\)](#),
[sigfillset\(\)](#),
[sigismember\(\)](#)

XBD [`<signal.h>`](#)

CHANGE HISTORY

首次发布于 Issue 3。

Issue 5

更新了 DESCRIPTION 以与 POSIX 线程扩展保持一致。

Issue 6

将 SYNOPSIS 标记为 CX，因为该函数在 [`<signal.h>`](#) 头文件中的存在是对 ISO C 标准的扩展。

1.216. sigprocmask

SYNOPSIS

```
#include <signal.h>

int pthread_sigmask(int how, const sigset_t *restrict set,
                     sigset_t *restrict oset);
int sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
```

DESCRIPTION

`pthread_sigmask()` 函数应检查或更改（或两者兼有）调用线程的信号掩码。

如果参数 `set` 不是空指针，它指向用于更改当前阻塞信号集的信号集。

参数 `how` 指示更改信号集的方式，应用程序应确保其由以下值之一组成：

`SIG_BLOCK`

结果集应为当前集与 `set` 指向的信号集的并集。

`SIG_SETMASK`

结果集应为 `set` 指向的信号集。

`SIG_UNBLOCK`

结果集应为当前集与 `set` 指向的信号集补集的交集。

如果参数 `oset` 不是空指针，则之前的掩码应存储在 `oset` 指向的位置。如果 `set` 是空指针，参数 `how` 的值不显著，线程的信号掩码应保持不变；因此该调用可用于查询当前被阻塞的信号。

如果参数 `set` 不是空指针，在 `pthread_sigmask()` 更改当前阻塞信号集后，它应确定是否有任何未决的非阻塞信号；如果有，则在 `pthread_sigmask()` 调用返回之前应至少传递这些信号中的一个。

无法阻塞那些不能被忽略的信号。系统应强制执行此规则而不导致指示错误。

如果在 SIGFPE、SIGILL、SIGSEGV 或 SIGBUS 信号被阻塞时生成了其中任何一个信号，结果未定义，除非该信号是由另一个进程的动作或由 `kill()`、`pthread_kill()`、`raise()` 或 `sigqueue()` 函数之一生成的。

如果 `pthread_sigmask()` 失败，线程的信号掩码不应更改。

`sigprocmask()` 函数应等价于 `pthread_sigmask()`，除了如果从多线程进程中调用其行为未指定，并且在出错时它返回 -1 并将 `errno` 设置为错误编号而不是直接返回错误编号。

RETURN VALUE

成功完成后，`pthread_sigmask()` 应返回 0；否则，它应返回相应的错误编号。

成功完成后，`sigprocmask()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

ERRORS

这些函数在以下情况失败：

[EINVAL]

`set` 参数不是空指针且 `how` 参数的值不等于定义的值之一。

这些函数不应返回 [EINTR] 错误代码。

EXAMPLES

多线程进程中的信号处理

此示例展示了 `pthread_sigmask()` 的使用，以便在多线程进程中处理信号。它提供了一个相当通用的框架，可以轻松地进行调整/扩展。

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <signal.h>
#include <string.h>
#include <errno.h>
...
static sigset_t    signal_mask; /* 要阻塞的信号          */
int main (int argc, char *argv[])
{
    pthread_t    sig_thr_id; /* 信号处理线程 ID      */
    int          rc;        /* 返回代码             */
    /* ... */
}
```

```
sigemptyset (&signal_mask);
sigaddset (&signal_mask, SIGINT);
sigaddset (&signal_mask, SIGTERM);
rc = pthread_sigmask (SIG_BLOCK, &signal_mask, NULL);
if (rc != 0) {
    /* 处理错误 */
    ...
}
/* 任何新创建的线程继承信号掩码 */

rc = pthread_create (&sig_thr_id, NULL, signal_thread, NULL);
if (rc != 0) {
    /* 处理错误 */
    ...
}
/* 应用程序代码 */
...
}

void *signal_thread (void *arg)
{
    int      sig_caught;      /* 捕获的信号 */
    int      rc;                /* 返回代码 */

    rc = sigwait (&signal_mask, &sig_caught);
    if (rc != 0) {
        /* 处理错误 */
    }
    switch (sig_caught)
    {
    case SIGINT:      /* 处理 SIGINT */
        ...
        break;
    case SIGTERM:     /* 处理 SIGTERM */
        ...
        break;
    default:          /* 通常不应该发生 */
        fprintf (stderr, "\n意外信号 %d\n", sig_caught);
        break;
    }
}
```

APPLICATION USAGE

虽然 `pthread_sigmask()` 在更改当前阻塞信号集后必须传递至少一个存在的未决非阻塞信号，但没有要求传递的信号包括那些被更改取消阻塞的信号。如果在 `pthread_sigmask()` 调用执行期间，一个或多个已经非阻塞的信号变为未决状态（参见 2.4.1 信号生成和传递），则在调用返回之前传递的信号可能仅包括那些信号。

RATIONALE

当在由 `sigaction()` 安装的信号捕获函数中更改线程的信号掩码时，从信号捕获函数返回时信号掩码的恢复会覆盖该更改（参见 `sigaction()`）。如果信号捕获函数是用 `signal()` 安装的，是否发生这种情况未指定。

关于信号传递要求的讨论，请参见 `kill()`。

FUTURE DIRECTIONS

无。

SEE ALSO

`exec` , `kill()` , `sigaction()` , `sigaddset()` , `sigdelset()` ,
`sigemptyset()` , `sigfillset()` , `sigismember()` , `sigpending()` ,
`sigqueue()` , `sigsuspend()`

XBD `<signal.h>`

CHANGE HISTORY

首次在 Issue 3 中发布。为了与 POSIX.1-1988 标准对齐而包含在内。

Issue 5

DESCRIPTION 已更新，以与 POSIX 线程扩展对齐。

`pthread_sigmask()` 函数已添加，以与 POSIX 线程扩展对齐。

Issue 6

`pthread_sigmask()` 函数被标记为 Threads 选项的一部分。

`sigprocmask()` 的 SYNOPSIS 被标记为 CX 扩展，以注意此函数在 `<signal.h>` 头中的存在是对 ISO C 标准的扩展。

为与 ISO POSIX-1:1996 标准对齐进行了以下更改：

- DESCRIPTION 已更新，以明确说明可能生成信号的函数。

规范文本已更新，以避免对应用程序要求使用术语"必须"。

`restrict` 关键字已添加到 `pthread_sigmask()` 和 `sigprocmask()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/105，将 DESCRIPTION 和 RATIONALE 部分中的"进程的信号掩码"更新为"线程的信号掩码"。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/106，向 EXAMPLES 部分添加了示例。

Issue 7

`pthread_sigmask()` 函数从 Threads 选项移动到 Base。

应用了 POSIX.1-2008，Technical Corrigendum 1，XSH/TC1-2008/0467 [319]。

Issue 8

应用了 Austin Group Defect 1132，澄清了 [EINVAL] 错误。

应用了 Austin Group Defect 1636，澄清了 `pthread_sigmask()` 和 `sigprocmask()` 等价性的例外情况。

应用了 Austin Group Defect 1731，澄清了虽然 `pthread_sigmask()` 在更改当前阻塞信号集后必须传递至少一个存在的未决非阻塞信号，但没有要求传递的信号包括那些被更改取消阻塞的信号。

1.217. sigqueue

概要 (SYNOPSIS)

```
#include <signal.h>

int sigqueue(pid_t pid, int signo, union sigval value);
```

描述 (DESCRIPTION)

`sigqueue()` 函数应将由 `signo` 指定的信号以及由 `value` 指定的值发送到由 `pid` 指定的进程。如果 `signo` 为零 (空信号)，则执行错误检查但实际上不发送信号。空信号可用于检查 `pid` 的有效性。

进程拥有向另一个进程队列发送信号的权限条件与 `kill()` 函数相同。

`sigqueue()` 函数应立即返回。如果为 `signo` 设置了 `SA_SIGINFO`，并且有资源可用于队列信号，则该信号应被排队并发送到接收进程。如果没有为 `signo` 设置 `SA_SIGINFO`，则 `signo` 应至少发送一次到接收进程；是否将 `value` 发送到接收进程为此调用的结果是不确定的。

如果 `pid` 的值导致为发送进程生成 `signo`，并且如果 `signo` 没有被调用线程阻塞，并且没有其他线程解除 `signo` 阻塞或在 `sigwait()` 函数中等待 `signo`，则在 `sigqueue()` 函数返回之前，`signo` 或至少挂起的、未阻塞的信号应传递给调用线程。如果选择了 `SIGRTMIN` 到 `SIGRTMAX` 范围内的多个挂起信号进行传递，应为编号最低的那个。实时信号和非实时信号之间的选择顺序，或多个挂起的非实时信号之间的选择顺序是不确定的。

返回值 (RETURN VALUE)

成功完成后，指定的信号应已被排队，并且 `sigqueue()` 函数应返回零值。否则，函数应返回 -1 值并设置 `errno` 以指示错误。

错误 (ERRORS)

`sigqueue()` 函数应在以下情况下失败：

- `[EAGAIN]`

- 没有资源可用于队列信号。进程已经排队了 {SIGQUEUE_MAX} 个在接收方处仍然挂起的信号，或者已超过系统范围的资源限制。
 - **[EINVAL]**
 - `signo` 参数的值是无效或不支持的信号编号。
 - **[EPERM]**
 - 进程没有适当的权限向接收进程发送信号。
 - **[ESRCH]**
 - 进程 `pid` 不存在。
-

以下部分为信息性内容。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

无。

基本原理 (RATIONALE)

`sigqueue()` 函数允许应用程序向自身或其他进程队列实时信号，指定应用程序定义的值。这在现有实时系统上的实时应用程序中是常见的做法。人们认为在已经为信号预留的 `sig` ... 名称空间中指定另一个函数，比扩展 `kill()` 的接口更可取。

当消息队列的 put/get 事件函数被移除时，这样的函数变得必要。应该注意的是，`sigqueue()` 函数在安全意识强的实现中意味着性能降低，因为发送方和接收方之间的访问权限必须在每次发送时检查，当 `pid` 被解析为目标进程时。在之前的接口中，此类访问检查仅在消息队列打开时是必要的。

标准开发者要求 `sigqueue()` 对于空信号具有与 `kill()` 相同的语义，并且使用相同的权限检查。但由于实现 `kill()` 的"广播"语义（例如，到进程组）的困难以及与资源分配的交互，没有采用这种语义。`sigqueue()` 函数将信号队列发送到由 `pid` 参数指定的单个进程。

如果系统没有足够的资源来队列信号, `sigqueue()` 函数可能失败。引入了进程可以发送的排队信号数量的明确限制。虽然限制是"每个发送方", 但 POSIX.1-2024 卷没有指定资源是发送方状态的一部分。这将要求发送方在退出后一直维护, 直到它发送到其他进程的所有信号都被处理, 或者所有尚未被处理的此类信号都从接收方的队列中移除。POSIX.1-2024 卷不排除此行为, 但也允许实现从系统范围的池分配排队资源 (具有每个发送方限制), 并且在发送方退出后保持排队信号挂起。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- [2.8.1 实时信号](#)
- XBD

变更历史 (CHANGE HISTORY)

首次发布于 Issue 5

为与 POSIX 实时扩展和 POSIX 线程扩展对齐而包含。

Issue 6

`sigqueue()` 函数被标记为实时信号扩展选项的一部分。
如果实现不支持实时信号扩展选项, 则移除了 [ENOSYS] 错误条件, 因为不需要提供存根。

Issue 7

`sigqueue()` 函数从实时信号扩展选项移动到基础。
POSIX.1-2008, 技术勘误 2, XSH/TC2-2008/0332 [844] 被应用。
信息性文本结束。

1.218. `sigsuspend` — 等待信号

概要

```
#include <signal.h>

int sigsuspend(const sigset_t *sigmask);
```

描述

`sigsuspend()` 函数应当原子性地执行以下两个操作：将调用线程的当前信号掩码替换为由 `sigmask` 指向的信号集，并挂起线程直到传递一个信号，该信号的动作是要执行信号捕获函数或终止进程。这不应导致可能在进程上挂起的任何其他信号在线程上变为挂起状态。

如果动作是终止进程，那么 `sigsuspend()` 永远不会返回。如果动作是执行信号捕获函数，那么 `sigsuspend()` 应在信号捕获函数返回后返回，并将信号掩码恢复到 `sigsuspend()` 调用之前存在的集合。

无法阻塞不能被忽略的信号。系统强制执行此规则而不会指示错误。

返回值

由于 `sigsuspend()` 无限期挂起线程执行，因此没有成功完成的返回值。如果发生返回，应返回 -1 并设置 `errno` 以指示错误。

错误

`sigsuspend()` 函数在以下情况下应失败：

- **[EINTR]** - 调用进程捕获了一个信号，并且控制从信号捕获函数返回。

应用程序用法

通常，在关键代码段的开始，使用 `sigprocmask()` 函数阻塞指定的信号集。当线程完成关键代码段并且需要等待先前阻塞的信号时，它通过调用 `sigsuspend()` 并使用 `sigprocmask()` 调用返回的掩码来暂停执行。

原理说明

希望避免线程取消处理器的信号掩码歧义的代码可以安装额外的取消处理器，将信号掩码重置为预期值。

```
void cleanup(void *arg)
{
    sigset_t *ss = (sigset_t *) arg;
    pthread_sigmask(SIG_SETMASK, ss, NULL);
}

int call_sigsuspend(const sigset_t *mask)
{
    sigset_t oldmask;
    int result;
    pthread_sigmask(SIG_SETMASK, NULL, &oldmask);
    pthread_cleanup_push(cleanup, &oldmask);
    result = sigsuspend(mask);
    pthread_cleanup_pop(0);
    return result;
}
```

参见

- [2.4 信号概念](#)
- [pause\(\)](#)
- [sigaction\(\)](#)
- [sigaddset\(\)](#)
- [sigdelset\(\)](#)
- [sigemptyset\(\)](#)
- [sigfillset\(\)](#)
-

变更历史

首次发布于 Issue 3

为与 POSIX.1-1988 标准对齐而包含。

Issue 5

更新了描述以与 POSIX 线程扩展对齐。

Issue 6

- 返回值部分中的文本已从 "suspends process execution" (挂起进程执行) 更改为 "suspends thread execution" (挂起线程执行)。这反映了 IEEE PASC 解释 1003.1c #40。
- 应用程序用法部分中的文本已被替换。
- 由于此函数在头文件中的存在是对 ISO C 标准的扩展，因此 SYNOPSIS 被标记为 CX。

Issue 7

应用了 SD5-XSH-ERN-122，在原理说明中添加了示例代码。

Issue 8

- 应用了 Austin Group 缺陷 1201，阐明了 `sigsuspend()` 的原子性要求。
 - 应用了 Austin Group 缺陷 1223，更改了原理说明中的示例代码。
-

1.219. `sigtimedwait`, `sigwaitinfo`

概要

```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
                  siginfo_t *restrict info,
                  const struct timespec *restrict timeout);

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

描述

`sigtimedwait()` 函数应等价于 `sigwaitinfo()`，不同之处在于如果由 `set` 指定的信号都没有挂起，`sigtimedwait()` 应等待由 `timeout` 引用的 **timespec** 结构中指定的时间间隔。如果由 `timeout` 指向的 **timespec** 结构为零值且由 `set` 指定的信号都没有挂起，那么 `sigtimedwait()` 应立即返回并带有错误。如果 `timeout` 是空指针，则行为未指定。应使用 `CLOCK_MONOTONIC` 时钟来测量由 `timeout` 参数指定的时间间隔。

`sigwaitinfo()` 函数从由 `set` 指定的集合中选择挂起的信号。如果在 `SIGRTMIN` 到 `SIGRTMAX` 范围内的多个挂起信号中有任何一个被选中，它应该是编号最小的那个。实时信号和非实时信号之间的选择顺序，或者多个挂起的非实时信号之间的选择顺序是未指定的。如果在调用时 `set` 中没有信号挂起，调用线程应被挂起，直到 `set` 中的一个或多个信号变为挂起或直到它被一个未阻塞的、已捕获的信号中断。

`sigwaitinfo()` 函数应等价于 `sigwait()` 函数，不同之处在于返回值和错误报告方法不同（参见返回值），并且如果 `info` 参数是非 `NULL`，选中的信号编号应存储在 `si_signo` 成员中，信号的原因应存储在 `si_code` 成员中。如果有任何值排队到选中的信号，第一个这样的排队值应被出队，并且如果 `info` 参数是非 `NULL`，该值应存储在 `info` 的 `si_value` 成员中。用于对信号进行排队的系统资源应被释放并返回给系统供其他用途。如果没有值排队，`si_value` 成员的内容是未定义的。如果没有进一步信号排队给选中的信号，该信号的挂起指示应被重置。

返回值

成功完成时（即，由 *set* 指定的信号之一是挂起的或已生成的），
`sigwaitinfo()` 和 `sigtimedwait()` 应返回选中的信号编号。否则，函数
应返回值 -1 并设置 *errno* 来指示错误。

错误

`sigtimedwait()` 函数应在以下情况失败：

- **[EAGAIN]**
- 在指定的超时期限内没有生成由 *set* 指定的信号。

`sigtimedwait()` 和 `sigwaitinfo()` 函数可能在以下情况失败：

- **[EINTR]**
- 等待被一个未阻塞的、已捕获的信号中断。是否此错误导致这些函数失败
应在系统文档中说明。

`sigtimedwait()` 函数也可能在以下情况失败：

- **[EINVAL]**
- *timeout* 参数指定的 *tv_nsec* 值小于零或大于或等于 1000 百万。

实现应该仅在 *set* 中没有信号挂起并且需要等待时才检查此错误。

应用用法

`sigtimedwait()` 函数超时并返回 [EAGAIN] 错误。应用程序开发者应注意，
这与其他函数如 `pthread_cond_timedwait()` 返回 [ETIMEDOUT] 不一致。

请注意，为确保生成的信号被排队且传递给 `sigqueue()` 的信号值在 *si_value*
中可用，使用 `sigwaitinfo()` 或 `sigtimedwait()` 的应用程序需要为集合
中的每个信号设置 SA_SIGINFO 标志（参见 2.4 信号概念）。这意味着将每个信号
设置为由三参数信号捕获函数处理，即使处理程序永远不会被调用。在设置
SA_SIGINFO 标志的同时将信号处理程序设置为 SIG_DFL 是不可能的（可移植地），
因为分配给 **struct sigaction** 的 *sa_handler* 成员而不是 *sa_sigaction* 成员会
导致未定义行为，并且 SIG_DFL 不必与 *sa_sigaction* 赋值兼容。即使编译器
接受将 SIG_DFL 赋值给 *sa_sigaction*，实现也不需要将此值视为特殊的——
它可能只是被当作信号捕获函数的地址。

基本原理

现有在实时系统上的编程实践使用暂停等待选定事件集的能力，并内联处理发生的一个事件，而不是在信号处理函数中。这允许应用程序以类似于状态机的事件驱动风格编写。这种编程风格对于大规模事务处理很有用，其中应用程序的整体吞吐量和清晰跟踪状态的能力比最小化单个事件处理的响应时间更重要。

可以从本 POSIX.1-2024 卷中定义的实时信号函数机制构造出一个信号等待宏函数。然而，这样的宏必须包括为所有要等待的信号定义的通用处理程序。如果信号等待函数由内核提供，则可以避免处理程序处理开销的很大一部分。因此，本 POSIX.1-2024 卷提供了两个信号等待函数——一个无限期等待，一个带超时——作为整体实时信号函数规范的一部分。

具有超时功能的函数规范允许编写一个应用程序，如果在设定的时间段内没有发生事件，可以中断等待。有人认为在等待之前设置定时器事件并在等待中识别定时器事件也会实现相同的功能，但性能水平较低。由于与用户级定时器事件规范相关联的性能下降，以及在等待完成有效事件后随后取消该定时器事件，以及与用户级方法相关联的处理潜在竞争条件的复杂性，包含了单独的函数。

请注意，`sigwaitinfo()` 函数的语义几乎与本 POSIX.1-2024 卷定义的 `sigwait()` 函数相同。唯一的区别是 `sigwaitinfo()` 在 `value` 参数中返回排队的信号值。返回排队值是必需的，以便应用程序可以区分排队到同一信号编号的多个事件。

维护这两个不同的函数是因为一些实现可能选择实现 POSIX 线程扩展函数而不实现排队信号扩展。但是请注意，如果 `value` 参数是 NULL，`sigwaitinfo()` 不返回排队值，因此 POSIX 线程扩展 `sigwait()` 函数可以在 `sigwaitinfo()` 上作为宏实现。

`sigtimedwait()` 函数从 `sigwaitinfo()` 函数中分离出来，以解决关于重载 `timeout` 指针以表示无限期等待（无超时）、定时等待和立即返回的担忧，以及关于与其他函数一致性的担忧，在这些函数中条件等待和定时等待是纯阻塞函数的独立函数。`sigtimedwait()` 的语义被指定为 `sigwaitinfo()` 可以用 `timeout` 的空指针作为宏实现。

`sigwait` 函数为线程提供了一种同步机制来等待异步生成的信号。一个重要问题是，当信号发送时，有多少在调用 `sigwait()` 函数等待信号的挂起线程应该从调用中返回。考虑了四种选择：

1. 对同一信号的多个同时调用 `sigwait` 函数返回错误。
2. 一个或多个线程返回。
3. 所有等待线程返回。

4. 恰好一个线程返回。

禁止对同一信号进行多次调用 `sigwait()` 被认为是过度限制的。"一个或多个"行为使符合规范的包实现容易，但代价是迫使 POSIX 线程客户端在应用程序代码中保护 `against` 多个同时调用 `sigwait()` 以实现可预测的行为。有人担心"所有等待线程"行为会导致"信号广播风暴"，通过在一般情况下复制信号来消耗过多的 CPU 资源。此外，无法提出有说服力的例子来说明传递给所有比传递给一个更简单或更强大。

因此，共识是应该恰好有一个在调用 `sigwait` 函数等待信号时挂起的线程在该信号发生时返回。这并不是一个繁重的限制，因为：

- 可以从单向等待构建多路信号等待。
- 信号应该只由应用程序级代码处理，因为库例程无法猜测应用程序想要对为整个进程生成的信号做什么。
- 因此应用程序可以安排单个线程等待任何给定信号，并在其到达时调用任何需要的例程。

在使用信号进行进程间通信的应用程序中，信号处理通常在一个地方完成。或者，如果信号被捕获以便可以进行进程清理，信号处理线程可以为应用程序的每个部分调用单独的进程清理例程。由于应用程序主线程启动了应用程序的每个部分，它处于正确的抽象级别来告诉应用程序的每个部分进行清理。

当然，存在一些编程风格，在多个线程中考虑等待单个信号是合乎逻辑的。可以构造一个简单的 `sigwait_multiple()` 例程来实现此目标。一个可能的实现是将每个 `sigwait_multiple()` 调用者注册为已表达对一组信号的兴趣。然后调用者在特定于线程的条件变量上等待。单个服务器线程在所有注册信号的并集上调用 `sigwait()` 函数。当 `sigwait()` 函数返回时，设置适当的状态并广播条件变量。新的 `sigwait_multiple()` 调用者可能导致挂起的 `sigwait()` 调用被取消并重新发出，以更新正在等待的信号集。

未来方向

无。

另见

2.4 信号概念，2.8.1 实时信号，`pause()`，`pthread_sigmask()`，`sigaction()`，`sigpending()`，`sigsuspend()`，`sigwait()`

XBD `<signal.h>`，`<time.h>`

变更历史

首次在 Issue 5 中发布。包含用于与 POSIX 实时扩展和 POSIX 线程扩展对齐。

Issue 6

这些函数被标记为实时信号扩展选项的一部分。

应用了 Open Group Corrigendum U035/3。 `sigwaitinfo()` 函数的概要已更正，使第二个参数为 `siginfo_t *` 类型。

[ENOSYS] 错误条件已被移除，因为如果实现不支持实时信号扩展选项，则不需要提供存根。

描述已更新以与 IEEE Std 1003.1j-2000 对齐，指定如果支持，则使用 CLOCK_MONOTONIC 时钟来测量超时间隔。

`restrict` 关键字已添加到 `sigtimedwait()` 和 `sigwaitinfo()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/130，将返回值部分中的措辞恢复到原始基础文档中的措辞（"实现应该仅在 `set` 中没有信号挂起并且需要等待时才检查此错误"）。

Issue 7

`sigtimedwait()` 和 `sigwaitinfo()` 函数已从实时信号扩展选项移动到基本标准。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0583 [392]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0333 [815]。

Issue 8

应用了 Austin Group Defect 1346，要求支持单调时钟。

1.220. `sigwait` — 等待排队的信号

概要

```
#include <signal.h>

int sigwait(const sigset_t *restrict set, int *restrict sig);
```

描述

`sigwait()` 函数应从 `set` 中选择一个挂起的信号，从系统的挂起信号集合中原子地清除它，并在 `sig` 引用的位置返回该信号编号。如果在调用 `sigwait()` 之前单个信号编号有多个挂起的实例，那么成功返回后该信号编号是否还有剩余挂起信号是由实现定义的。如果实现支持排队信号，并且为所选信号编号排队了多个信号，则第一个此类排队信号应导致 `sigwait()` 返回，其余信号应保持排队状态。如果在调用时 `set` 中没有信号挂起，线程应被挂起，直到一个或多个信号变为挂起状态。`set` 定义的信号在调用 `sigwait()` 时应已被阻塞；否则，行为是未定义的。`sigwait()` 对 `set` 中信号的信号动作的影响是未指定的。

如果多个线程使用 `sigwait()` 等待相同的信号，这些线程中最多只有一个应从 `sigwait()` 返回信号编号。如果超过一个线程在 `sigwait()` 中为某个信号而被阻塞，当该信号为进程生成时，哪个等待线程从 `sigwait()` 返回是未指定的。如果信号为特定线程生成，如通过 `pthread_kill()`，则只有该线程应返回。

如果选择了 SIGRTMIN 到 SIGRTMAX 范围内的多个挂起信号中的任何一个，它应该是编号最低的那个。实时信号和非实时信号之间的选择顺序，或者多个挂起的非实时信号之间的选择顺序是未指定的。

返回值

成功完成时，`sigwait()` 应将接收到信号的信号编号存储在 `sig` 引用的位置并返回零。否则，应返回错误编号以指示错误。

错误

`sigwait()` 函数可能会失败：

- [EINVAL] `set` 参数包含无效或不支持的信号编号。

示例

无。

应用用法

无。

基本原理

为线程提供等待信号的便捷方式，POSIX.1-2024 本卷提供了 `sigwait()` 函数。对于线程必须等待信号的大多数情况，`sigwait()` 函数应该相当方便、高效和足够。

然而，有请求要求提供比 `sigwait()` 更底层的原语以及线程可以使用的信号量。经过一些考虑后，允许线程使用信号量，并且 `sem_post()` 被定义为异步信号安全的。

总之，当需要在响应异步信号运行的代码通知线程时，应使用 `sigwait()` 来处理信号。或者，如果实现提供信号量，它们也可以使用，可以跟在 `sigwait()` 之后使用，或者在之前通过 `sigaction()` 注册的信号处理例程内部使用。

未来方向

无。

参见

- [信号概念](#)
- [实时信号](#)

- `pause()`
- `pthread_sigmask()`
- `sigaction()`
- `sigpending()`
- `sigsuspend()`
- `sigtimedwait()`
- `<signal.h>`
- `<time.h>`

变更历史

首次在 Issue 5 中发布。为与 POSIX 实时扩展和 POSIX 线程扩展保持一致而包含。

Issue 6

为与 ISO/IEC 9899:1999 标准保持一致，向 `sigwait()` 原型添加了 `restrict` 关键字。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/131，更新描述以说明如果超过一个线程在 `sigwait()` 中被阻塞，哪个等待线程返回是未指定的，并且如果信号为特定线程生成，只有该线程应返回。

Issue 7

与实时信号扩展选项相关的功能被移至基础规范。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0584 [76]。

1.221. `sigtimedwait`, `sigwaitinfo` — 等待排队信号

SYNOPSIS (函数概要)

```
#include <signal.h>

int sigtimedwait(const sigset_t *restrict set,
                  siginfo_t *restrict info,
                  const struct timespec *restrict timeout);

int sigwaitinfo(const sigset_t *restrict set,
                siginfo_t *restrict info);
```

DESCRIPTION (描述)

`sigtimedwait()` 函数应等同于 `sigwaitinfo()`，不同之处在于：如果 `set` 指定的信号都没有挂起，`sigtimedwait()` 应等待 `timeout` 引用的 `timespec` 结构体中指定的时间间隔。如果 `timeout` 指向的 `timespec` 结构体为零值且 `set` 指定的信号都没有挂起，那么 `sigtimedwait()` 应立即返回并带有错误。如果 `timeout` 是空指针，则行为未指定。应使用 CLOCK_MONOTONIC 时钟来测量 `timeout` 参数指定的时间间隔。

`sigwaitinfo()` 函数从 `set` 指定的集合中选择挂起的信号。如果在 SIGRTMIN 到 SIGRTMAX 范围内有多个挂起信号被选中，应为编号最小的那个。实时信号和非实时信号之间，或多个挂起的非实时信号之间的选择顺序是未指定的。如果在调用时 `set` 中没有信号挂起，调用线程应被挂起，直到 `set` 中的一个或多个信号变为挂起，或者被一个未阻塞的、已捕获的信号中断。

`sigwaitinfo()` 函数应等同于 `sigwait()` 函数，不同之处在于返回值和错误报告方法不同（参见 RETURN VALUE），并且如果 `info` 参数非空，选中的信号编号应存储在 `si_signo` 成员中，信号的原因应存储在 `si_code` 成员中。如果有任何值排队到选中的信号，第一个这样的排队值应被出队，并且如果 `info` 参数非空，该值应存储在 `info` 的 `si_value` 成员中。用于排队信号的系统资源应被释放并返回给系统供其他用途。如果没有值排队，`si_value` 成员的内容是未定义的。如果没有进一步信号排队到选中的信号，该信号的挂起指示应被重置。

RETURN VALUE (返回值)

成功完成时 (即 `set` 指定的信号之一是挂起的或生成的), `sigwaitinfo()` 和 `sigtimedwait()` 应返回选中的信号编号。否则, 函数应返回值 -1 并设置 `errno` 以指示错误。

ERRORS (错误)

`sigtimedwait()` 函数可能在以下情况失败:

- **[EAGAIN]**
- 在指定的超时期限内没有生成 `set` 指定的信号。

`sigtimedwait()` 和 `sigwaitinfo()` 函数可能在以下情况失败:

- **[EINTR]**
- 等待被未阻塞的、已捕获的信号中断。应在系统文档中记录此错误是否导致这些函数失败。

`sigtimedwait()` 函数也可能在以下情况失败:

- **[EINVAL]**
- `timeout` 参数指定的 `tv_nsec` 值小于零或大于等于 10 亿。

实现应仅在 `set` 中没有信号挂起且需要等待时才检查此错误。

APPLICATION USAGE (应用程序使用)

`sigtimedwait()` 函数超时并返回 [EAGAIN] 错误。应用程序开发者应注意, 这与其他函数 (如返回 [ETIMEDOUT] 的 `pthread_cond_timedwait()`) 不一致。

请注意, 为确保生成的信号被排队且传递给 `sigqueue()` 的信号值在 `si_value` 中可用, 使用 `sigwaitinfo()` 或 `sigtimedwait()` 的应用程序需要为集合中的每个信号设置 SA_SIGINFO 标志 (参见 2.4 信号概念)。这意味着将每个信号设置为由三参数信号捕获函数处理, 即使处理程序永远不会被调用。在设置 SA_SIGINFO 标志的同时 (可移植地) 将信号处理程序设置为 SIG_DFL 是不可能的, 因为分配给 `struct sigaction` 的 `sa_handler` 成员而不是 `sa_sigaction` 成员会导致未定义行为, 并且 SIG_DFL 不需要与 `sa_sigaction` 赋值兼容。即使编译器接受将 SIG_DFL 赋值给

`sa_sigaction`，实现也不需要将此值视为特殊——它可能只是被当作信号捕获函数的地址。

RATIONALE (基本原理)

实时系统上现有的编程实践使用暂停等待选定事件集并在行内处理发生的第一 个事件（而不是在信号处理函数中）的能力。这允许应用程序以事件导向的样 式编写，类似于状态机。这种编码风格对于大规模事务处理很有用，其中应用 程序的总体吞吐量和清晰跟踪状态的能力比最小化单个事件处理的响应时间更 重要。

可以从 POSIX.1-2024 中定义的实时信号函数机制构造一个信号等待宏函数。但 是，这样的宏必须包括所有要等待信号的广义处理程序的定义。如果信号等待 函数由内核提供，可以避免处理程序处理开销的很大一部分。因此，POSIX.1- 2024 提供了两个信号等待函数——一个无限等待，一个带超时——作为整体实 时信号函数规范的一部分。

具有超时的函数规范允许编写一个应用程序，如果没有事件发生，可以在设 定的时间段后中断等待。有人认为在等待之前设置定时器事件并在等待中识别定 时器事件也会实现相同的功能，但性能水平较低。由于与用户级定时器事件规 范相关的性能下降，以及等待为有效事件完成后该定时器事件的后续取消，以 及与用户级方法相关的潜在竞争条件处理的复杂性，包含了单独的函数。

请注 意，`sigwaitinfo()` 函数的语义几乎与 POSIX.1-2024 定义的 `sigwait()` 函数相同。唯一的区别是 `sigwaitinfo()` 在 `value` 参数中返 回排队的信号值。返回排队值是必需的，以便应用程序可以区分排队到同一信 号编号的多个事件。

保留两个不同的函数是因为一些实现可能选择实现 POSIX 线程扩展函数而不实 现排队信号扩展。但请注意，如果 `value` 参数为 NULL，`sigwaitinfo()` 不会返 回排队值，所以 POSIX 线程扩展 `sigwait()` 函数可以作为 `sigwaitinfo()` 的宏实现。

将 `sigtimedwait()` 函数从 `sigwaitinfo()` 函数分离是为了解决关于 `timeout` 指针的重载以指示无限等待（无超时）、定时等待和立即返回的担 忧，以及关于与条件等待和定时等待是独立于纯阻塞函数的其他函数一致性的 担忧。`sigtimedwait()` 的语义被指定为 `sigwaitinfo()` 可以作为使用空 指针作为 `timeout` 的宏实现。

`sigwait` 函数为线程提供了同步机制以等待异步生成的信号。一个重要问题 是：当发送信号时，有多少个在对同一信号的 `sigwait()` 函数调用中挂起的 线程应该从调用中返回。考虑了四个选择：

1. 对同一信号的多次同时 `sigwait` 函数调用返回错误。

2. 一个或多个线程返回。
3. 所有等待线程返回。
4. 正好一个线程返回。

禁止对同一信号多次调用 `sigwait()` 被认为过于限制性。"一个或多个"行为使实现符合规范的包变得容易，但代价是强迫 POSIX 线程客户端在应用程序代码中保护免受对 `sigwait()` 的多次同时调用以实现可预测的行为。有担忧认为"所有等待线程"行为会导致"信号广播风暴"，通过在一般情况下的信号复制消耗过多的 CPU 资源。此外，无法提出有说服力的例子说明传递给所有比传递给一个更简单或更强大。

因此，共识是在对信号的 `sigwait` 函数调用中挂起的正好一个线程应该在信号发生时返回。这不是一个繁重的限制，因为：

- 可以从单向等待构建多路信号等待。
- 信号只应由应用程序级代码处理，因为库例程无法猜测应用程序想要对为整个进程生成的信号做什么。
- 因此应用程序可以安排单个线程等待任何给定信号并在其到达时调用任何需要的例程。

在使用信号进行进程间通信的应用程序中，信号处理通常在一个地方完成。或者，如果信号被捕获以便进行进程清理，信号处理线程可以为应用程序的每个部分调用单独的进程清理例程。由于应用程序主代码启动了应用程序的每个部分，它处于正确的抽象级别来告诉应用程序的每个部分进行清理。

当然，存在一些编程风格，在多个线程中等待单个信号是合乎逻辑的。可以构造一个简单的 `sigwait_multiple()` 例程来实现这个目标。一个可能的实现是让每个 `sigwait_multiple()` 调用者注册为表示对一组信号感兴趣。然后调用者在线程特定的条件变量上等待。单个服务器线程在所有注册信号的并集上调用 `sigwait()` 函数。当 `sigwait()` 函数返回时，设置适当的状态并广播条件变量。新的 `sigwait_multiple()` 调用者可能导致挂起的 `sigwait()` 调用被取消并重新发出，以更新正在等待的信号集。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (另见)

- 2.4 信号概念

- 2.8.1 实时信号

- `pause()`
- `pthread_sigmask()`
- `sigaction()`
- `sigpending()`
- `sigsuspend()`
- `sigwait()`
- `<signal.h>`
- `<time.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 5

为与 POSIX 实时扩展和 POSIX 线程扩展对齐而包含。

Issue 6

- 这些函数被标记为实时信号扩展选项的一部分。
- 应用了 Open Group 更正 U035/3。 `sigwaitinfo()` 函数的 SYNOPSIS 已更正，使第二个参数为 `siginfo_t *` 类型。
- [ENOSYS] 错误条件已被删除，因为如果实现不支持实时信号扩展选项，不需要提供存根。
- 描述已为与 IEEE Std 1003.1j-2000 对齐而更新，指定如果支持 CLOCK_MONOTONIC 时钟，则用于测量超时间隔。
- `restrict` 关键字已添加到 `sigtimedwait()` 和 `sigwaitinfo()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。
- 应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/130，将 RETURN VALUE 部分的措辞恢复到原始基础文档中的内容（"实现应仅在 set 中没有信号挂起且需要等待时才检查此错误"）。

Issue 7

- `sigtimedwait()` 和 `sigwaitinfo()` 函数已从实时信号扩展选项移动到基本规范。
- 应用了 POSIX.1-2008, 技术更正 1, XSH/TC1-2008/0583 [392]。
- 应用了 POSIX.1-2008, 技术更正 2, XSH/TC2-2008/0333 [815]。

Issue 8

- 应用了 Austin Group 缺陷 1346, 要求支持单调时钟。

1.222. snprintf, asprintf, dprintf, fprintf, printf, sprintf — 打印格式化输出

概要

```
#include <stdio.h>

[CX] int asprintf(char **restrict ptr, const char *restrict for
[CX] int dprintf(int fildes, const char *restrict format, ...);
int fprintf(FILE *restrict stream, const char *restrict format,
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n,
             const char *restrict format, ...);
int sprintf(char *restrict s, const char *restrict format, ...)
```

描述

[CX] 除了 `asprintf()`、`dprintf()` 以及传递空宽字符时 `%lc` 转换的行为外，本参考页描述的功能与 ISO C 标准一致。此处描述的要求与 ISO C 标准之间的任何其他冲突都是无意的。POSIX.1-2024 的这一卷在 `fprintf()`、`printf()`、`snprintf()` 和 `sprintf()` 功能方面遵循 ISO C 标准，但在传递空宽字符时的 `%lc` 转换方面除外。

`fprintf()` 函数应将输出放置在指定的输出流 `stream` 上。`printf()` 函数应将输出放置在标准输出流 `stdout` 上。`sprintf()` 函数应将输出后跟空字节 '\0' 放置在从 `*s` 开始的连续字节中；确保有足够的空间可用是用户的责任。

[CX] `asprintf()` 函数应等价于 `sprintf()`，不同的是输出字符串应写入动态分配的内存，该内存分配方式如同调用 `malloc()`，长度应足以容纳结果字符串，包括终止空字节。如果 `asprintf()` 调用成功，此动态分配字符串的地址应存储在 `ptr` 引用的位置。

`dprintf()` 函数应等价于 `fprintf()` 函数，不同的是 `dprintf()` 应将输出写入与 `fildes` 参数指定的文件描述符关联的文件，而不是将输出放置在流上。

`snprintf()` 函数应等价于 `sprintf()`，但增加了 `n` 参数，该参数限制写入 `s` 引用的缓冲区的字节数。如果 `n` 为零，则不应写入任何内容，且 `s` 可

能为空指针。否则，超出第 `n-1` 个字节的输出字节将被丢弃而不是写入数组，并在实际写入数组的字节末尾写入一个空字节。

如果由于调用 `sprintf()` 或 `snprintf()` 导致重叠的对象之间发生复制，则结果未定义。

这些函数中的每一个都在 `format` 的控制下转换、格式化和打印其参数。应用程序应确保格式是字符字符串，如果存在任何移位状态，则以初始移位状态开始和结束。格式由零个或多个指令组成：普通字符（仅复制到输出流）和转换规范（每个都导致获取零个或多个参数）。如果格式没有足够的参数，则结果未定义。如果格式用尽而参数仍有剩余，则多余参数将被求值但除此之外被忽略。

[CX] 转换可以应用于参数列表中 `format` 之后的第 n 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 %（见下文）被序列 "% n \$" 替换，其中 n 是范围 [1,{NL_ARGMAX}] 中的十进制整数，给出参数在参数列表中的位置。此功能提供了定义以适合特定语言的顺序选择参数的格式字符串（见示例部分）。

`format` 可以包含编号参数转换规范（即以 "% n \$" 引入并可选择性包含 "* m \$" 形式的字段宽度和精度）或非编号参数转换规范（即以 % 字符引入并可选择性包含 * 形式的字段宽度和精度），但不能同时包含两者。唯一的例外是 %% 可以与 "% n \$" 形式混合使用。在格式字符串中混合编号和非编号参数规范的结果是未定义的。使用编号参数规范时，指定第 N 个参数要求所有前导参数（从第一个到第 $(N-1)$ 个）都在格式字符串中指定。

在包含 "% n \$" 形式转换规范的格式字符串中，参数列表中的编号参数可以根据需要从格式字符串中引用多次。

在包含 % 形式转换规范的格式字符串中，每个转换规范使用参数列表中第一个未使用的参数。

[CX] 所有形式的 `fprintf()` 函数都允许在输出字符串中插入依赖于语言的小数点字符。小数点字符在当前语言环境（类别 `LC_NUMERIC`）中定义。在 POSIX 语言环境中，或在小数点字符未定义的语言环境中，小数点字符应默认为 ('.')。

每个转换规范由 '%' 字符 [CX] 或字符序列 "% n \$" 引入，其后按顺序出现以下内容：

- 零个或多个标志（按任意顺序），它们修改转换规范的含义。
- 可选的最小字段宽度。如果转换后的值具有比字段宽度更少的字节，默认情况下应在左侧用字符填充；如果给字段宽度指定左对齐标志（'-'）（如下所述），则应在右侧填充。字段宽度采用 ('') 的形式，[CX] 或在以 "% n \$" 引入的转换规范中采用 "m\$" 字符串（如下所述），或十进制整数的形式。

- 可选的**精度**，它给出 d、i、o、u、x 和 X 转换说明符要显示的最小数字位数；a、A、e、E、f 和 F 转换说明符的小数点后要显示的数字位数；g 和 G 转换说明符的最大有效数字位数；或者要从 s [XSI] 和 S 转换说明符中的字符串打印的最大字节数。精度采用 ('.') 后跟 ('') 的形式，[CX] 或在以 "%n\$" 引入的转换规范中采用 "m\$" 字符串（如下所述），或可选的十进制数字字符串的形式，其中空数字字符串被视为零。如果精度与任何其他转换说明符一起出现，则行为未定义。
- 可选的**长度修饰符**，指定参数的大小。
- **转换说明符字符**，指示要应用的转换类型。

字段宽度、精度或两者都可以用 ('') 表示。在这种情况下，`int` 类型的参数提供字段宽度或精度。应用程序应确保指定字段宽度、精度或两者的参数按此顺序出现在要转换的参数（如果有）之前。负字段宽度被视为 '-' 标志后跟正字段宽度。负精度被视为省略精度。[CX] 在包含以 "%n\$" 引入的转换规范的格式字符串中，除了用十进制数字字符串指示外，字段宽度可以用序列 "m\$" 表示，精度可以用序列 ".m\$" 表示，其中 m 是范围 [1,{NL_ARGMAX}] 中的十进制整数，给出包含字段宽度或精度的整数参数在参数列表中的位置（在 `format` 参数之后），例如：

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

标志字符

标志字符及其含义是：

' (撇号)

[CX] 十进制转换（%i、%d、%u、%f、%F、%g 或 %G）结果的整数部分应使用千位分组字符格式化。对于其他转换，行为未定义。使用非货币分组字符。

- 转换的结果应在字段内左对齐。如果未指定此标志，则转换右对齐。

+

有符号转换的结果应始终以符号（'+' 或 '-'）开始。如果未指定此标志，则仅当转换负值时转换才以符号开始。

如果有符号转换的第一个字符不是符号，或者有符号转换的结果没有字符，则应在结果前缀。这意味着如果 和 '+' 标志都出现，则应忽略 标志。

#

指定值要转换为替代形式。对于 o 转换，它应增加精度，当且仅当必要时，以

强制结果的第一个数字为零（如果值和精度都为 0，则打印单个 0）。对于 x 或 X 转换说明符，非零结果应前缀 0x（或 0X）。对于 a、A、e、E、f、F、g 和 G 转换说明符，结果应始终包含小数点字符，即使小数点字符后没有数字。没有此标志，仅当小数点字符后跟数字时，这些转换的结果中才会出现小数点字符。对于 g 和 G 转换说明符，尾随零不应从结果中删除，就像它们通常被删除一样。对于其他转换说明符，行为未定义。

0

对于 d、i、o、u、x、X、a、A、e、E、f、F、g 和 G 转换说明符，前导零（在任何符号或基数指示之后）用于填充到字段宽度，而不是执行空格填充，转换无穷大或 NaN 时除外。如果 '0' 和 '-' 标志都出现，则忽略 '0' 标志。对于 d、i、o、u、x 和 X 转换说明符，如果指定了精度，则应忽略 '0' 标志。[CX] 如果 '0' 和 '-' 标志都出现，则分组字符在零填充之前插入。对于其他转换，行为未定义。

长度修饰符

长度修饰符及其含义是：

hh

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `signed char` 或 `unsigned char` 参数（参数将根据整数提升进行提升，但其值应在打印前转换为 `signed char` 或 `unsigned char`）；或者后续的 n 转换说明符应用于指向 `signed char` 参数的指针。

h

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `short` 或 `unsigned short` 参数（参数将根据整数提升进行提升，但其值应在打印前转换为 `short` 或 `unsigned short`）；或者后续的 n 转换说明符应用于指向 `short` 参数的指针。

l (ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `long` 或 `unsigned long` 参数；后续的 n 转换说明符应用于指向 `long` 参数的指针；后续的 c 转换说明符应用于 `wint_t` 参数；后续的 s 转换说明符应用于指向 `wchar_t` 参数的指针；或者对后续的 a、A、e、E、f、F、g 或 G 转换说明符没有影响。

ll (ell-ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `long long` 或 `unsigned long long` 参数；或者后续的 n 转换说明符应用于指向 `long long` 参数的指针。

j

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `intmax_t` 或 `uintmax_t`

参数；或者后续的 n 转换说明符应用于指向 `intmax_t` 参数的指针。

z

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `size_t` 或相应的有符号整数类型参数；或者后续的 n 转换说明符应用于指向对应于 `size_t` 参数的有符号整数类型的指针。

t

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `ptrdiff_t` 或相应的 `unsigned` 类型参数；或者后续的 n 转换说明符应用于指向 `ptrdiff_t` 参数的指针。

L

指定后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于 `long double` 参数。

如果长度修饰符与除上述指定之外的任何转换说明符一起出现，则行为未定义。

转换说明符

转换说明符及其含义是：

d, i

`int` 参数应转换为 "[-]dddd" 样式的有符号十进制数。精度指定要显示的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

o

`unsigned` 参数应转换为 "dddd" 样式的无符号八进制数。精度指定要显示的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

u

`unsigned` 参数应转换为 "dddd" 样式的无符号十进制数。精度指定要显示的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

x

`unsigned` 参数应转换为 "dddd" 样式的无符号十六进制数；使用字母 "abcdef"。精度指定要显示的最小数位数；如果要转换的值可以用更少的位数表示，则应用前导零扩展。默认精度为 1。用显式精度零转换零的结果应为无字符。

X

等价于 x 转换说明符，但使用字母 "ABCDEF" 而不是 "abcdef"。

f, F

`double` 参数应转换为 "[-]ddd.ddd" 样式的十进制记数法，其中小数点字符后的数位数等于精度规范。如果缺少精度，则应取为 6；如果精度显式为零且没有 '#' 标志，则不应出现小数点字符。如果出现小数点字符，则它前面至少出现一个数字。低位数字应以实现定义的方式四舍五入。

表示无穷大的 `double` 参数应转换为 "[-]inf" 或 "[-]infinity" 样式之一；使用哪种样式是实现定义的。表示 NaN 的 `double` 参数应转换为 "[-]nan(n-char-sequence)" 或 "[-]nan" 样式之一；使用哪种样式以及任何 n-char-sequence 的含义是实现定义的。F 转换说明符产生 "INF"、"INFINITY" 或 "NAN"，而不是 "inf"、"infinity" 或 "nan"。

e, E

`double` 参数应转换为 "[-]d.ddd_e±dd" 样式，其中小数点字符前有一个数字（如果参数非零则非零），其后的数位数等于精度；如果缺少精度，则应取为 6；如果精度为零且没有 '#' 标志，则不应出现小数点字符。低位数字应以实现定义的方式四舍五入。E 转换说明符应产生用 'E' 而不是 'e' 引入指数的数字。指数应始终包含至少两位数字。如果值为零，则指数应为零。

表示无穷大或 NaN 的 `double` 参数应以 f 或 F 转换说明符的样式转换。

g, G

表示浮点数的 `double` 参数应根据转换的值和精度以 f 或 e 样式转换（或者在 G 转换说明符的情况下以 F 或 E 样式转换）。令 P 等于精度（如果非零），省略精度时为 6，或者精度为零时为 1。那么，如果使用 E 样式的转换的指数为 X：

- 如果 $P > X \geq -4$ ，则转换应使用 f（或 F）样式和精度 $P-(X+1)$ 。
- 否则，转换应使用 e（或 E）样式和精度 P-1。

最后，除非使用 '#' 标志，否则应从结果的小数部分中删除任何尾随零，如果没有剩余的小数部分，则应删除小数点字符。

表示无穷大或 NaN 的 `double` 参数应以 f 或 F 转换说明符的样式转换。

a, A

表示浮点数的 `double` 参数应转换为 "[-]0x_h._hhhh_p±d" 样式，其中小数点字符前有一个十六进制数字（如果参数是归一化浮点数则非零，否则未指定），其后的十六进制数位数等于精度；如果缺少精度且 FLT_RADIX 是 2 的幂，则精度应足以精确表示值；如果缺少精度且 FLT_RADIX 不是 2 的幂，则精度应足以区分 `double` 类型的值，但可以省略尾随零；如果精度为零且未指定 '#' 标志，则不应出现小数点字符。a 转换使用字母 "abcdef"，A 转换使用字母 "ABCDEF"。A 转换说明符产生用 'X' 和 'P' 而不是 'x' 和 'p' 的数字。指数应始终包含至少一位数字，并且仅包含表示 2 的十进制指数所需的更多数字。如果值为零，则指数应为零。

表示无穷大或 NaN 的 `double` 参数应以 `f` 或 `F` 转换说明符的样式转换。

c

`int` 参数应转换为 `unsigned char`，结果字节应被写入。

如果存在 `l` (ell) 限定符，[CX] `wint_t` 参数应转换为多字节序列，如同通过调用 `wcrtomb()` 并使用指向至少 `MB_CUR_MAX` 字节的存储指针、转换为 `wchar_t` 的 `wint_t` 参数以及初始移位状态，并写入结果字节。

s

参数应是指向 `char` 数组的指针。应从数组中写入字节，直到（但不包括）任何终止空字节。如果指定了精度，则写入的字节数不应超过该数量。如果未指定精度或精度大于数组大小，应用程序应确保数组包含空字节。

如果存在 `l` (ell) 限定符，参数应是指向 `wchar_t` 类型数组的指针。应将数组中的宽字符转换为字符（每个都如同通过调用 `wcrtomb()` 函数，转换状态由在第一个宽字符转换前初始化为零的 `mbstate_t` 对象描述），直到并包括终止空宽字符。应写入结果字符，直到（但不包括）终止空字符（字节）。如果未指定精度，应用程序应确保数组包含空宽字符。如果指定了精度，则写入的字符（字节）数不应超过该数量（包括任何移位序列），并且如果为了等于精度给定的字符序列长度，函数需要访问数组末尾之后的宽字符，则数组应包含空宽字符。在任何情况下都不应写入部分字符。

p

参数应是指向 `void` 的指针。指针的值以实现定义的方式转换为可打印字符序列。

n

参数应是指向整数的指针，其中写入通过调用 `fprintf()` 函数之一到目前为止写入输出的字节数。不转换任何参数。

c

[XSI] 等价于 `lc`。

s

[XSI] 等价于 `ls`。

%

写入 '%' 字符；不转换任何参数。应用程序应确保完整的转换规范是 %%。

如果转换规范与上述形式之一不匹配，则行为未定义。如果任何参数不是相应转换规范的正确类型，则行为未定义。

在任何情况下，不存在或小的字段宽度都不会导致字段截断；如果转换的结果比字段宽度宽，则应扩展字段以包含转换结果。`fprintf()` 和 `printf()` 生成的字符被打印，如同调用了 `fputc()`。

对于 a 和 A 转换说明符，如果 FLT_RADIX 是 2 的幂，则值应正确舍入为具有给定精度的十六进制浮点数。

对于 a 和 A 转换，如果 FLT_RADIX 不是 2 的幂且结果在给定精度下不能精确表示，则结果应该是给定精度的十六进制浮点样式中的两个相邻数字之一，额外要求误差对于当前舍入方向应有正确符号。

对于 e、E、f、F、g 和 G 转换说明符，如果有效十进制数字的数量最多为 DECIMAL_DIG，则结果应正确舍入。如果有效十进制数字的数量超过 DECIMAL_DIG 但源值可以用 DECIMAL_DIG 位数字精确表示，则结果应该是带尾随零的精确表示。否则，源值由两个相邻的十进制字符串 L < U 限定，两者都具有 DECIMAL_DIG 个有效数字；结果十进制字符串 D 的值应满足 L <= D <= U，额外要求误差对于当前舍入方向应有正确符号。

[CX] 文件的最后数据修改和最后文件状态更改时间戳应标记为更新：

1. 在调用 `fprintf()` 或 `printf()` 成功执行与在同一流上成功完成调用 `fflush()` 或 `fclose()` 或调用 `exit()` 或 `abort()` 之间
2. 在 `dprintf()` 调用成功完成时

返回值

成功完成后，[CX] `dprintf()`、`fprintf()` 和 `printf()` 函数应返回传输的字节数。

[CX] 成功完成后，`asprintf()` 函数应返回写入存储在 `ptr` 引用位置的已分配字符串的字节数，不包括终止空字节。

成功完成后，`sprintf()` 函数应返回写入 `s` 的字节数，不包括终止空字节。

成功完成后，`snprintf()` 函数应返回如果 `n` 足够大时将写入 `s` 的字节数，不包括终止空字节。

如果遇到错误，这些函数应返回负值 [CX] 并设置 `errno` 以指示错误。对于 `asprintf()`，如果内存分配不可能，或者发生其他错误，函数应返回负值，`ptr` 引用的位置内容未定义，但不应引用已分配的内存。

如果在调用 `snprintf()` 时 `n` 的值为零，则不应写入任何内容，应返回如果 `n` 足够大时将要写入的字节数（不包括终止空字节），并且 `s` 可能为空指针。

错误

有关 [CX] `dprintf()`、`fprintf()` 和 `printf()` 失败和可能失败的条件，请参考 `fputc()` 或 `fputwc()`。

此外，所有形式的 `fprintf()` 在以下情况下将失败：

[EILSEQ]

[CX] 检测到不对应于有效字符的宽字符代码。

[EOVERFLOW]

[CX] 要返回的值大于 {INT_MAX}。

[CX] `asprintf()` 函数在以下情况下将失败：

[ENOMEM]

存储空间不足。

`dprintf()` 函数在以下情况下可能失败：

[EBADF]

`fildes` 参数不是有效的文件描述符。

[CX] `dprintf()`、`fprintf()` 和 `printf()` 函数在以下情况下可能失败：

[ENOMEM]

[CX] 存储空间不足。

示例

打印语言无关的日期和时间

以下语句可用于使用语言无关格式打印日期和时间：

```
printf(format, weekday, month, day, hour, min);
```

对于美国用法，`format` 可以是指向以下字符串的指针：

```
"%s, %s %d, %d:%.2d\n"
```

此示例将产生以下消息：

```
Sunday, July 3, 10:02
```

对于德国用法，`format` 可以是指向以下字符串的指针：

```
"%1$s, %3$d. %2$s, %4$d:%5$.2d\n"
```

此 `format` 定义将产生以下消息：

```
Sonntag, 3. Juli, 10:02
```

打印文件信息

以下示例打印目录中特定文件的类型、权限和链接数信息。

```
#include <stdio.h>
#include <sys/types.h>
#include <pwd.h>
#include <grp.h>

char *strperm (mode_t);
...
struct stat statbuf;
struct passwd *pwd;
struct group *grp;
...
printf("%10.10s", strperm (statbuf.st_mode));
printf("%4d", statbuf.st_nlink);

if ((pwd = getpwuid(statbuf.st_uid)) != NULL)
    printf(" %-8.8s", pwd->pw_name);
else
    printf(" %-8ld", (long) statbuf.st_uid);

if ((grp = getgrgid(statbuf.st_gid)) != NULL)
    printf(" %-8.8s", grp->gr_name);
else
    printf(" %-8ld", (long) statbuf.st_gid);

printf("%9jd", (intmax_t) statbuf.st_size);
```

打印本地化日期字符串

以下示例获取本地化日期字符串：

```
#include <stdio.h>
#include <time.h>
#include <langinfo.h>
...
struct dirent *dp;
```

```
struct tm *tm;
char datestring[256];
...
strftime(datestring, sizeof(datestring), nl_langinfo (D_T_FMT),
printf(" %s %s\n", datestring, dp->d_name);
```

打印错误信息

以下示例使用 `fprintf()` 将错误信息写入标准错误：

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

#define LOCKFILE "/etc/ptmp"
#define PASSWDFILE "/etc/passwd"

...
int pfd;
...
if ((pfd = open(LOCKFILE, O_WRONLY | O_CREAT | O_EXCL,
    S_IRUSR | S_IWUSR | S_IRGRP | S_IROTH)) == -1)
{
    fprintf(stderr, "Cannot open /etc/ptmp. Try again later.\n"
    exit(1);
}
...
if (link(LOCKFILE,PASSWDFILE) == -1) {
    fprintf(stderr, "Link error: %s\n", strerror(errno));
    exit(1);
}
```

打印用法信息

以下示例检查程序是否有必要的参数，如果没有预期的参数数量，则使用 `fprintf()` 打印用法信息：

```
#include <stdio.h>
#include <stdlib.h>
...
```

```
char *Options = "hdbtl";
...
if (argc < 2) {
    fprintf(stderr, "Usage: %s -%s <file\n", argv[0], Options);
}
```

格式化十进制字符串

以下示例在 `stdout` 上打印键和数据对。注意格式字符串中 ('*') 的使用；这确保根据请求的元素数量为元素提供正确的小数位数：

```
#include <stdio.h>
...
long i;
char *keystr;
int elementlen, len;
...
while (len < elementlen) {
...
    printf("%s Element%0*ld\n", keystr, elementlen, i);
...
}
```

创建路径名

以下示例使用先前返回用户密码数据库条目的 `getpwnam()` 函数的信息创建路径名：

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <unistd.h>
...
char *pathname;
struct passwd *pw;
size_t len;
...
// pid_t 所需的位数是位数乘以
// log2(10) = 约 10/33
len = strlen(pw->pw_dir) + 1 + 1+(sizeof(pid_t)*80+32)/33 +
      sizeof ".out";
pathname = malloc(len);
if (pathname != NULL)
```

```
{  
    snprintf(pathname, len, "%s/%jd.out", pw->pw_dir,  
             (intmax_t)getpid());  
    ...  
}
```

报告事件

以下示例循环直到事件超时。`pause()` 函数除非接收到信号，否则永远等待。由于 `pause()` 的可能返回值，`fprintf()` 语句永远不应该出现：

```
#include <stdio.h>  
#include <unistd.h>  
#include <string.h>  
#include <errno.h>  
...  
while (!event_complete) {  
    ...  
    if (pause() != -1 || errno != EINTR)  
        fprintf(stderr, "pause: unknown error: %s\n", strerror(  
    }  
◀ ▶
```

打印货币信息

以下示例使用 `strfmon()` 转换数字并将其存储为名为 `convbuf` 的格式化货币字符串。如果打印第一个数字，程序打印格式和描述；否则，它只打印数字：

```
#include <monetary.h>  
#include <stdio.h>  
...  
struct tblfmt {  
    char *format;  
    char *description;  
};  
  
struct tblfmt table[] = {  
    { "%n", "default formatting" },  
    { "%11n", "right align within an 11 character field" },  
    { "%#5n", "aligned columns for values up to 99999" },  
    { "%-*#5n", "specify a fill character" },  
    { "%=0#5n", "fill characters do not use grouping" },  
    { "%^#5n", "disable the grouping separator" },  
    { "%^#5.0n", "round off to whole units" },  
    { "%^#5.4n", "increase the precision" },
```

```

    { "%(#5n", "use an alternative pos/neg style" },
    { "%!(#5n", "disable the currency symbol" },
};

...
float input[3];
int i, j;
char convbuf[100];
...

strfmon(convbuf, sizeof(convbuf), table[i].format, input[j]);

if (j == 0) {
    printf("%s %s %s\n", table[i].format,
           convbuf, table[i].description);
}
else {
    printf(" %s\n", convbuf);
}

```

打印宽字符

以下示例打印一系列宽字符。假设 "L@@" 扩展为三个字节：

```

wchar_t wz [3] = L"@@";           // 零终止
wchar_t wn [3] = L"@@";           // 非终止

fprintf (stdout,"%ls", wz);      // 输出 6 字节
fprintf (stdout,"%ls", wn);      // 未定义, 因为 wn 没有终止符
fprintf (stdout,"%4ls", wz);     // 输出 3 字节
fprintf (stdout,"%4ls", wn);     // 输出 3 字节; 不需要终止符
fprintf (stdout,"%9ls", wz);     // 输出 6 字节
fprintf (stdout,"%9ls", wn);     // 输出 9 字节; 不需要终止符
fprintf (stdout,"%10ls", wz);    // 输出 6 字节
fprintf (stdout,"%10ls", wn);    // 未定义, 因为 wn 没有终止符

```

在示例的最后一行中，处理三个字符后，已输出九个字节。然后必须检查第四个字符以确定它是转换为一个字节还是更多字节。如果它转换为多个字节，则输出只有九个字节。由于数组中没有第四个字符，行为未定义。

应用程序用法

如果调用 `fprintf()` 的应用程序具有任何 `wint_t` 或 `wchar_t` 类型的对象，它还必须包含 `<wchar.h>` 头文件以定义这些对象。

成功调用 `asprintf()` 分配的空间应随后通过调用 `free()` 释放。

基本原理

如果实现检测到格式没有足够的参数，建议函数应失败并报告 [EINVAL] 错误。

为 %lc 转换指定的行为与 ISO C 标准中的规范略有不同，打印空宽字符产生空字节而不是 0 字节输出，如同对 ISO C 标准指示应用 %ls 说明符到第一个元素为空宽字符的 `wchar_t` 数组的严格读法所要求的那样。要求每个可能的宽字符（包括空字符）都有多字节输出符合历史实践，并与 `fprintf()` 中的 %c 以及 `fwprintf()` 中的 %c 和 %lc 提供一致性。预期 ISO C 标准的未来版本将更改以匹配此处指定的行为。

未来方向

无。

另请参见

- 2.5 标准 I/O 流
- `fputc()`
- `fscanf()`
- `setlocale()`
- `strfmon()`
- `strlcat()`
- `wcrtomb()`
- `wcslcat()`

XBD 7. 语言环境, `<inttypes.h>`, `<stdio.h>`, `<wchar.h>`

更改历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

与 ISO/IEC 9899:1990/Amendment 1:1995 (E) 对齐。具体来说，l (ell) 限定符现在可以与 c 和 s 转换说明符一起使用。

`snprintf()` 函数在 Issue 5 中是新的。

Issue 6

超出 ISO C 标准的扩展被标记。

规范文本更新，避免对应用程序要求使用术语"必须"。

为与 ISO/IEC 9899:1999 标准对齐进行以下更改：

- `fprintf()`、`printf()`、`snprintf()` 和 `sprintf()` 的原型更新，并从 `snprintf()` 中移除 XSI 阴影。
- `snprintf()` 的描述与 ISO C 标准对齐。注意，这取代了 Open Group Base Resolution bwg98-006 中的 `snprintf()` 描述，该描述改变了 Issue 5 的行为。
- 描述更新。

描述更新，一致使用术语"转换说明符"和"转换规范"。

合并 ISO/IEC 9899:1999 标准，技术勘误表 1。

添加打印宽字符的示例。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #161，更新 0 标志的描述。

应用 Austin Group Interpretation 1003.1-2001 #170。

应用 ISO/IEC 9899:1999 标准，技术勘误表 2 #68 (SD5-XSH-ERN-70)，修订 g 和 G 的描述。

应用 SD5-XSH-ERN-174。

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 添加 `dprintf()` 函数。

与 %n\$ 形式转换规范和 标志相关的功能从 XSI 选项移至 Base。

进行与支持细粒度时间戳相关的更改。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0163 [302], XSH/TC1-2008/0164 [316], XSH/TC1-2008/0165 [316], XSH/TC1-2008/0166 [451,291], 和 XSH/TC1-2008/0167 [14]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0126 [894], XSH/TC2-2008/0127 [557], 和 XSH/TC2-2008/0128 [936]。

Issue 8

应用 Austin Group Defect 986，将 `strlcat()` 和 `wcslcat()` 添加到另请参见部分。

应用 Austin Group Defect 1020，阐明 `snprintf()` 参数 `n` 限制写入 `s` 的字节数；它不一定与 `s` 的大小相同。

应用 Austin Group Defect 1021，将返回值部分中的"输出错误"更改为"错误"。

应用 Austin Group Defect 1137，阐明在转换规范中使用 "%n\$" 和 "*m\$"。

应用 Austin Group Defect 1205，更改 % 转换说明符的描述。

应用 Austin Group Defect 1219，删除 `snprintf()` 特定的 [EOVERFLOW] 错误。

应用 Austin Group Defect 1496，添加 `asprintf()` 函数。

应用 Austin Group Defect 1562，阐明确保格式是字符字符串（如果存在任何移位状态，则以初始移位状态开始和结束）是应用程序的责任。

应用 Austin Group Defect 1647，更改 c 转换说明符的描述，并更新本卷 POSIX.1-2024 遵循 ISO C 标准的语句，以便它排除传递空宽字符时的 %lc 转换。

1.223. sprintf - 打印格式化输出

SYNOPSIS

```
#include <stdio.h>

int asprintf(char **restrict ptr, const char *restrict format,
int dprintf(int fildes, const char *restrict format, ...);

int fprintf(FILE *restrict stream, const char *restrict format,
int printf(const char *restrict format, ...);
int snprintf(char *restrict s, size_t n, const char *restrict f
int sprintf(char *restrict s, const char *restrict format, ...)
```

DESCRIPTION

`fprintf()` 函数应将输出放置在指定的输出流 `stream` 上。`printf()` 函数应将输出放置在标准输出流 `stdout` 上。`sprintf()` 函数应将输出后跟空字节 '\0' 放置在从 `s` 开始的连续字节中；用户有责任确保有足够的空间可用。

`asprintf()` 函数应等价于 `sprintf()`，不同之处在于输出字符串应写入动态分配的内存，该内存通过调用 `malloc()` 分配，长度足够容纳结果字符串，包括终止空字节。如果对 `asprintf()` 的调用成功，则此动态分配字符串的地址应存储在由 `ptr` 引用的位置。

`dprintf()` 函数应等价于 `fprintf()` 函数，不同之处在于 `dprintf()` 应将输出写入与 `fildes` 参数指定的文件描述符关联的文件，而不是将输出放置在流上。

`snprintf()` 函数应等价于 `sprintf()`，增加了 `n` 参数，该参数限制写入到由 `s` 引用的缓冲区的字节数。如果 `n` 为零，则不应写入任何内容，且 `s` 可能为空指针。否则，超过第 `n-1` 个的输出字节将被丢弃而不是写入数组，并且在实际写入数组的字节末尾写入一个空字节。

如果由于调用 `sprintf()` 或 `snprintf()` 而在重叠的对象之间进行复制，则结果未定义。

这些函数中的每一个都在 `format` 的控制下转换、格式化并打印其参数。应用程序应确保格式是一个字符字符串，如果有，则以其初始移位状态开始和结束。格式由零个或多个指令组成：普通字符，这些字符简单地复制到输出流；和转换说明，每个转换说明应导致获取零个或多个参数。如果格式没有足够的

参数，则结果未定义。如果格式已用尽而参数仍有剩余，则多余的参数应被求值但被忽略。

转换可以应用于参数列表中格式之后的第 n 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 %（见下文）被序列 "% n \$" 替换，其中 n 是范围 [1,{NL_ARGMAX}] 中的十进制整数，给出参数在参数列表中的位置。此功能提供了定义格式字符串的功能，该字符串以适合特定语言的顺序选择参数（参见示例部分）。

格式可以包含编号参数转换说明（即，由 "% n \$" 引入并可选地包含字段宽度和精度的 "* m \$" 形式），或未编号参数转换说明（即，由 % 字符引入并可选地包含字段宽度和精度的 * 形式），但不能同时包含两者。唯一的例外是 %% 可以与 "% n \$" 形式混合。在格式字符串中混合编号和未编号参数说明的结果是未定义的。当使用编号参数说明时，指定第 N 个参数需要所有前导参数（从第一个到第 $(N-1)$ 个）都在格式字符串中指定。

在包含 "% n \$" 形式转换说明的格式字符串中，参数列表中的编号参数可以从格式字符串中按需引用多次。

在包含 % 形式转换说明的格式字符串中，每个转换说明使用参数列表中的第一个未使用参数。

所有形式的 `fprintf()` 函数都允许在输出字符串中插入依赖于语言的小数字符。小数字符在当前语言环境（类别 LC_NUMERIC）中定义。在 POSIX 语言环境中，或者在小数字符未定义的语言环境中，小数字符应默认为句点（'.'）。

每个转换说明由 '%' 字符或字符序列 "% n \$" 引入，之后按顺序出现以下内容：

- 零个或多个标志（按任意顺序），这些标志修改转换说明的含义。
- 一个可选的最小字段宽度。如果转换后的值比字段宽度有更少的字节，则默认应在左侧用空格字符填充；如果给定了左对齐标志（'-'），则应在右侧填充。字段宽度采用星号（'*」的形式，或者在由 "% n \$" 引入的转换说明中采用下面描述的 " m \$" 字符串，或者采用十进制整数。
- 一个可选的精度，该精度给出 d、i、o、u、x 和 X 转换说明符要出现的最小小数位数；a、A、e、E、f 和 F 转换说明符的小数字符后要出现的数字位数；g 和 G 转换说明符的最大有效数字位数；或 s 转换说明符要从字符串打印的最大字节数。精度采用句点（'.'）后跟星号（'*」的形式，或者在由 "% n \$" 引入的转换说明中采用下面描述的 " m \$" 字符串，或者采用可选的十进制数字字符串，其中空数字字符串被视为零。如果精度与任何其他转换说明符一起出现，则行为未定义。
- 一个可选的长度修饰符，用于指定参数的大小。
- 一个转换说明符字符，指示要应用的转换类型。

字段宽度、精度或两者都可以用星号 ('*) 表示。在这种情况下，`int` 类型的参数提供字段宽度或精度。应用程序应确保指定字段宽度、精度或两者的参数在被转换的参数（如果有）之前按该顺序出现。负字段宽度被视为 '-' 标志后跟正字段宽度。负精度被视为省略了精度。在包含由 "%n\$" 引入的转换说明的格式字符串中，除了用十进制数字字符串表示外，字段宽度可以用序列 "m\$" 表示，精度用序列 ".m\$" 表示，其中 m 是范围 [1,{NL_ARGMAX}] 中的十进制整数，给出包含字段宽度或精度的整数参数在参数列表中的位置（在格式参数之后），例如：

```
printf("%1$d:%2$.*3$d:%4$.*3$d\n", hour, min, precision, sec);
```

标志字符

标志字符及其含义是：

' (撇号)

十进制转换（%i、%d、%u、%f、%F、%g 或 %G）结果的整数部分应使用千位分组字符进行格式化。对于其他转换，行为未定义。使用非货币分组字符。

- 转换结果应在字段内左对齐。如果未指定此标志，则转换右对齐。

+

有符号转换的结果应始终以符号（'+' 或 '-'）开始。如果未指定此标志，则仅当转换负值时才以符号开始。

空格

如果有符号转换的第一个字符不是符号，或者有符号转换没有产生字符，则应在结果前添加空格。这意味着如果空格和 '+' 标志都出现，则忽略空格标志。

#

指定值要转换为替代形式。对于 o 转换，它应在必要时增加精度，以强制结果的第一个数字为零（如果值和精度都为 0，则打印单个 0）。对于 x 或 X 转换说明符，非零结果应前缀 0x（或 0X）。对于 a、A、e、E、f、F、g 和 G 转换说明符，结果应始终包含小数字符，即使小数字符后没有数字。没有此标志，仅当小数字符后跟数字时，这些转换的结果中才会出现小数字符。对于 g 和 G 转换说明符，尾随零不应像通常那样从结果中移除。对于其他转换说明符，行为未定义。

0

对于 d、i、o、u、x、X、a、A、e、E、f、F、g 和 G 转换说明符，使用前导零（跟在任何符号或基数指示之后）来填充到字段宽度，而不是执行空格填充，除非转换无穷大或 NaN。如果 '0' 和 '-' 标志都出现，则忽略 '0' 标志。对于 d、

i、o、u、x 和 X 转换说明符，如果指定了精度，则忽略 '0' 标志。如果 '0' 和撇号标志都出现，则在零填充之前插入分组字符。对于其他转换，行为未定义。

长度修饰符

长度修饰符及其含义是：

hh

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `signed char` 或 `unsigned char` 参数（参数将根据整数提升进行提升，但其值应在打印前转换为 `signed char` 或 `unsigned char`）；或指定后续的 n 转换说明符应用于指向 `signed char` 参数的指针。

h

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `short` 或 `unsigned short` 参数（参数将根据整数提升进行提升，但其值应在打印前转换为 `short` 或 `unsigned short`）；或指定后续的 n 转换说明符应用于指向 `short` 参数的指针。

l (ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `long` 或 `unsigned long` 参数；指定后续的 n 转换说明符应用于指向 `long` 参数的指针；指定后续的 c 转换说明符应用于 `wint_t` 参数；指定后续的 s 转换说明符应用于指向 `wchar_t` 参数的指针；或对后续的 a、A、e、E、f、F、g 或 G 转换说明符没有影响。

ll (ell-ell)

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `long long` 或 `unsigned long long` 参数；或指定后续的 n 转换说明符应用于指向 `long long` 参数的指针。

j

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `intmax_t` 或 `uintmax_t` 参数；或指定后续的 n 转换说明符应用于指向 `intmax_t` 参数的指针。

z

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `size_t` 或相应的有符号整数类型参数；或指定后续的 n 转换说明符应用于指向对应于 `size_t` 参数的有符号整数类型的指针。

t

指定后续的 d、i、o、u、x 或 X 转换说明符应用于 `ptrdiff_t` 或相应的无符号类型参数；或指定后续的 n 转换说明符应用于指向 `ptrdiff_t` 参数的指针。

L

指定后续的 a、A、e、E、f、F、g 或 G 转换说明符应用于 `long double` 参数。

如果长度修饰符与除上述指定之外的任何转换说明符一起出现，则行为未定义。

转换说明符

转换说明符及其含义是：

d, i

`int` 参数应转换为 "[-]dddd" 样式的有符号十进制。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应使用前导零扩展。默认精度为 1。将零转换为显式精度为零的结果应为无字符。

o

`unsigned` 参数应转换为 "dddd" 样式的无符号八进制格式。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应使用前导零扩展。默认精度为 1。将零转换为显式精度为零的结果应为无字符。

u

`unsigned` 参数应转换为 "dddd" 样式的无符号十进制格式。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应使用前导零扩展。默认精度为 1。将零转换为显式精度为零的结果应为无字符。

x

`unsigned` 参数应转换为 "dddd" 样式的无符号十六进制格式；使用字母 "abcdef"。精度指定要出现的最小数位数；如果要转换的值可以用更少的位数表示，则应使用前导零扩展。默认精度为 1。将零转换为显式精度为零的结果应为无字符。

X

等价于 x 转换说明符，但使用字母 "ABCDEF" 而不是 "abcdef"。

f, F

`double` 参数应转换为 "[-]ddd.ddd" 样式的十进制表示法，其中小数字符后的数位数等于精度规范。如果缺少精度，则应取为 6；如果精度显式为零且没有 '#' 标志，则不应出现小数字符。如果出现小数字符，则在其前面至少出现一个数字。低位数字应以实现定义的方式四舍五入。

表示无穷大的 `double` 参数应以 "[-]inf" 或 "[-]infinity" 样式之一转换；使用哪种样式是实现定义的。表示 NaN 的 `double` 参数应以 "[-]nan(n-char-sequence)" 或 "[-]nan" 样式之一转换；使用哪种样式以及任何 n-char-

sequence 的含义是实现定义的。F 转换说明符分别产生 "INF"、"INFINITY" 或 "NAN"，而不是 "inf"、"infinity" 或 "nan"。

e, E

`double` 参数应转换为 "[-]d.ddd e \pm dd" 样式，其中小数字符前有一个数字（如果参数非零则非零），其后的数字位数等于精度；如果缺少精度，则应取为 6；如果精度为零且没有 '#' 标志，则不应出现小数字符。低位数字应以实现定义的方式四舍五入。E 转换说明符应产生以 'E' 而不是 'e' 引入指数的数字。指数应始终包含至少两位数字。如果值为零，则指数应为零。

表示无穷大或 NaN 的 `double` 参数应以 f 或 F 转换说明符的样式转换。

g, G

表示浮点数的 `double` 参数应根据转换的值和精度以 f 或 e 样式（或在 G 转换说明符的情况下以 F 或 E 样式）转换。设 P 等于精度（如果非零）、省略精度时为 6，或精度为零时为 1。那么，如果使用样式 E 的转换将具有指数 X：

- 如果 $P > X \geq -4$ ，则转换应使用样式 f（或 F）和精度 $P-(X+1)$ 。
- 否则，转换应使用样式 e（或 E）和精度 P-1。

最后，除非使用 '#' 标志，否则应从结果的小数部分移除任何尾随零，如果没有剩余的小数部分，则移除小数点字符。

表示无穷大或 NaN 的 `double` 参数应以 f 或 F 转换说明符的样式转换。

a, A

表示浮点数的 `double` 参数应转换为 "[-]0x h.ffffp \pm d" 样式，其中小数字符前有一个十六进制数字（如果参数是归一化浮点数则非零，否则未指定），其后的十六进制数字位数等于精度；如果缺少精度，则应取为足够表示浮点数的精确值，如果精度为零且没有 '#' 标志，则不应出现小数字符。a 转换使用字母 "abcdef"，A 转换使用字母 "ABCDEF"。A 转换说明符产生使用 'X' 和 'P' 而不是 'x' 和 'p' 的数字。指数应始终包含至少一个数字，且仅包含表示 2 的十进制指数所需的更多数字。

表示无穷大或 NaN 的 `double` 参数应以 f 或 F 转换说明符的样式转换。

c

如果没有 l 长度修饰符，则 `int` 参数应转换为 `unsigned char`，结果字节应被写入。

如果有 l 长度修饰符，则 `wint_t` 参数应如同通过调用 `wcrtomb()` 且没有状态一样转换，结果的多字节字符应被写入。

s

如果没有 l 长度修饰符，应用程序应确保参数是指向字符数组的指针，该数组包含以初始移位状态开始的字符序列。应写入数组中的字符直到（但不包括）终止空字节。如果指定了精度，则不应写入超过该数量的字节，并且不应写入任

何部分多字节字符。如果未指定精度或精度大于数组的大小，应用程序应确保数组包含空字节。

如果有 `l` 长度修饰符，应用程序应确保参数是指向 `wchar_t` 类型数组的指针。数组中的宽字符应转换为多字节字符（每个如同通过调用 `wcrtomb()` 转换，转换状态由在第一个宽字符转换之前初始化为零的 `mbstate_t` 对象描述），结果的多字节字符应被写入直到（但不包括）终止空宽字符。如果指定了精度，则不应写入超过该数量的字节，并且不应写入任何部分多字节字符。如果未指定精度或精度大于转换后的宽字符序列的大小，应用程序应确保数组包含空宽字符。

p

应用程序应确保参数是指向 `void` 的指针。指针的值以实现定义的方式转换为可打印字符序列。

n

应用程序应确保参数是指向整数的指针，该整数中写入此调用到 `fprintf()` 函数之一到目前为止写入输出流的字节数。不转换任何参数。格式字符串应按原样写入输出；如果转换说明包含字段宽度或精度，则写入输出的字节数的值应如同指定了这些值一样确定。

对于 `snprintf()`，这意味着如果 `n` 足够大，则本应写入缓冲区 `s` 的字节数，不包括终止空字节。

%

应写入 '%' 字符。不转换任何参数。完整的转换说明应为 "%%"。

如果转换说明不匹配上述形式之一，则行为未定义。

RETURN VALUE

成功完成后，`fprintf()`、`printf()`、`dprintf()`、`snprintf()`、`asprintf()` 和 `sprintf()` 函数应返回传输的字节数。

如果遇到输出错误，这些函数应返回负值。

如果在调用 `snprintf()` 时 `n` 的值为零，则不应写入任何内容，应返回如果 `n` 足够大本应写入的字节数，且 `s` 可能为空指针。

ERRORS

如果出现以下情况，函数将失败：

EILSEQ

在宽字符转换中检测到非法字节序列。

EOVERFLOW

要存储的值大于相应类型的整数。

`fprintf()`、`printf()`、`dprintf()`、`snprintf()` 和 `sprintf()` 函数在以下情况下可能失败：

ENOMEM

可用的存储空间不足。

`snprintf()` 函数在以下情况下将失败：

EOVERFLOW

`n` 的值大于 `{INT_MAX}` 或提供的存储空间不足。

`asprintf()` 函数在以下情况下将失败：

ENOMEM

可用的内存空间不足以存储结果字符串。

`dprintf()` 函数在以下情况下将失败：

EBADF

流底层的文件描述符不是有效的文件描述符或未打开用于写入。

`fprintf()` 和 `printf()` 函数在以下情况下可能失败：

ENOMEM

可用的存储空间不足。

EILSEQ

检测到不对应有效字符的宽字符代码。

EXAMPLES

示例 1：打印与语言无关的日期和时间

以下语句可用于以与语言无关的方式打印日期和时间：

```
printf (format, weekday, month, day, hour, min);
```

对于美国用法，`format` 可以是指向以下字符串的指针：

```
"%s, %s %d, %.2d:%.2d\n"
```

此示例可能产生以下输出：

Sunday, July 3, 10:02

对于德国用法, `format` 可以是指向以下字符串的指针:

```
"%1$s, %3$d. %2$s, %4$02.2d:%5$02.2d\n"
```

此 `format` 定义可能产生以下输出:

```
Sonntag, 3. Juli, 10:02
```

示例 2: 打印文件列表

要以 Sunday, July 3, 10:02 的形式打印日期和时间, 其中 `weekday` 和 `month` 是指向以空字符结尾的字符串的指针:

```
printf("%s, %s %d, %.2d:%.2d\n", weekday, month, day, hour, min
```

要将 `pi` 打印到 5 位小数:

```
printf("pi = %.5f\n", 3.1415926535);
```

上述示例可能产生以下输出:

```
pi = 3.14159
```

示例 3: 打印到不同目标

```
#include <stdio.h>
#include <stdlib.h>

int main(void) {
    char buffer[100];
    char *dynamic_buffer = NULL;
    int ret;

    /* 写入标准输出 */
    ret = printf("Hello, World!\n");

    /* 写入缓冲区 */
    ret = sprintf(buffer, "Count: %d", 42);

    /* 写入带大小限制的缓冲区 */
}
```

```
ret = snprintf(buffer, sizeof(buffer), "Value: %f", 3.14159

/* 写入动态分配的缓冲区 */
ret = asprintf(&dynamic_buffer, "String: %s", "Dynamic");
if (ret != -1) {
    printf("Allocated: %s\n", dynamic_buffer);
    free(dynamic_buffer);
}

return 0;
}
```

APPLICATION USAGE

在调用 `asprintf()` 之前, `*ptr` 的值是不确定的, 不应假定为 `NULL`。如果调用成功, 应用程序应使用 `free()` 释放由 `asprintf()` 分配的内存。

RATIONALE

`sprintf()` 函数容易出现缓冲区溢出。创建 `snprintf()` 函数是为了通过限制写入缓冲区的字节数来解决此安全问题。

添加 `asprintf()` 函数是为了处理程序员事先不知道格式化输出需要多大缓冲区的常见情况。▶

编号参数说明 ("%n\$") 为词序与英语不同的语言提供支持, 允许重新排序格式字符串而不更改函数调用中的参数顺序。

FUTURE DIRECTIONS

无。

SEE ALSO

`fclose()`, `fopen()`, `fputc()`, `puts()`, `scanf()`, `wcrtomb()`

1.224. rand, srand — 伪随机数生成器

SYNOPSIS

```
#include <stdlib.h>

int rand(void);
void srand(unsigned seed);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`rand()` 函数应计算 $[0, \{RAND_MAX\}]$ [XSI] 范围内的伪随机整数序列，周期至少为 2^{32} 。

`rand()` 函数不需要是线程安全的；但是，`rand()` 应避免与非线程安全的伪随机序列生成函数之外的所有函数发生数据竞争。

`srand()` 函数使用参数作为新的伪随机数序列的种子，该序列将由后续对 `rand()` 的调用返回。如果随后使用相同的种子值调用 `srand()`，则应重复伪随机数序列。如果在任何对 `srand()` 的调用之前调用 `rand()`，则将生成与首次使用种子值 1 调用 `srand()` 时相同的序列。

`srand()` 函数不需要是线程安全的；但是，`srand()` 应避免与非线程安全的伪随机序列生成函数之外的所有函数发生数据竞争。

实现的行为应如同 POSIX.1-2024 本卷中定义的任何函数都不调用 `rand()` 或 `srand()`。

RETURN VALUE

`rand()` 函数应返回序列中的下一个伪随机数。

`srand()` 函数不应返回值。

ERRORS

未定义错误。

EXAMPLES

生成伪随机数序列

以下示例演示如何生成伪随机数序列。

```
#include <stdio.h>
#include <stdlib.h>
...
long count, i;
char *keystr;
int elementlen, len;
char c;
...
/* 初始化随机数生成器。 */
srand(1);

/* 仅使用小写字符创建键 */
len = 0;
for (i=0; i<count; i++) {
    while (len < elementlen) {
        c = (char) (rand() % 128);
        if (islower(c))
            keystr[len++] = c;
    }

    keystr[len] = '\0';
    printf("%s Element%0*ld\n", keystr, elementlen, i);
    len = 0;
}
```

在不同机器上生成相同序列

以下代码定义了一对函数，可以集成到希望确保在不同机器上生成相同数字序列的应用程序中。

```
static unsigned long next = 1;

int myrand(void) /* 假设 RAND_MAX 为 32767。 */
{
    next = next * 1103515245 + 12345;
    return((unsigned)(next/65536) % 32768);
}

void mysrand(unsigned seed)
{
```

```
    next = seed;  
}
```

APPLICATION USAGE

当需要满足非平凡要求（包括安全性）时，应避免使用这些函数，除非使用 `getentropy()` 进行种子初始化。

`drand48()` 和 `random()` 函数提供了更为精密的伪随机数生成器。

RATIONALE

ISO C 标准的 `rand()` 和 `srand()` 函数允许所有线程共享每个进程的伪随机流。这两个函数不需要更改，但必须有互斥机制防止两个线程同时访问随机数生成器时的干扰。

关于 `rand()`，在多线程程序中可能需要两种不同的行为：

1. 所有调用 `rand()` 的线程共享的单个每进程伪随机数序列
2. 每个调用 `rand()` 的线程的不同伪随机数序列

这是通过修改的线程安全函数提供的，该函数基于种子值是对整个进程全局还是对每个线程本地。

这并未解决 `rand()` 函数实现中已知的缺陷，这些缺陷已通过维护更多状态来处理。实际上，这指定了有缺陷函数的新线程安全形式。

FUTURE DIRECTIONS

无。

SEE ALSO

- `drand48()`
- `getentropy()`
- `initstate()`
- `<stdlib.h>`

CHANGE HISTORY

首次发布于 Issue 1

源自 SVID 的 Issue 1。

Issue 5

- 包含 `rand_r()` 函数以与 POSIX 线程扩展保持一致。
- 在 DESCRIPTION 中添加了说明 `rand()` 函数不需要是可重入的注释。

Issue 6

- 标记了超出 ISO C 标准的扩展。
- 将 `rand_r()` 函数标记为线程安全函数选项的一部分。

Issue 7

- 应用了 Austin Group Interpretation 1003.1-2001 #156。
- 将 `rand_r()` 函数标记为过时。
- 应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0301 [743]。

Issue 8

- 应用了 Austin Group Defect 1134，添加了 `getentropy()`。
 - 应用了 Austin Group Defect 1302，将这些函数与 ISO/IEC 9899:2018 标准保持一致。
 - 应用了 Austin Group Defect 1330，移除了过时接口。
-

1.225. fscanf, scanf, sscanf

概要

```
#include <stdio.h>

int fscanf(FILE *restrict stream, const char *restrict format,
int scanf(const char *restrict format, ...);
int sscanf(const char *restrict s, const char *restrict format,
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 标准遵循 ISO C 标准。

`fscanf()` 函数应从指定的输入 `stream` 读取。`scanf()` 函数应从标准输入流 `stdin` 读取。`sscanf()` 函数应从字符串 `s` 读取。每个函数读取字节，根据格式解释它们，并将结果存储在其参数中。每个函数期望作为参数：一个如下描述的控制字符串 `format`，以及一组指示转换后的输入应存储位置的 `pointer` 参数。如果格式的参数不足，结果是未定义的。如果格式用尽而参数仍有剩余，多余的参数应被求值，但在其他情况下被忽略。

[CX] 转换可以应用于参数列表中 `format` 之后的第 `n` 个参数，而不是下一个未使用的参数。在这种情况下，转换说明符字符 `%`（见下文）被序列 `"%n$"` 替换，其中 `n` 是范围 `[1,{NL_ARGMAX}]` 内的十进制整数。此功能提供了定义格式字符串的能力，该字符串以适合特定语言的顺序选择参数。在包含 `"%n$"` 形式转换说明的格式字符串中，参数列表中的编号参数是否可以从格式字符串中多次引用是未指定的。

格式可以包含转换说明的任何一种形式——即 `%` 或 `"%n$"`——但两种形式不能在单个格式字符串中混合使用。唯一的例外是 `%%` 或 `%*` 可以与 `"%n$"` 形式混合使用。当使用编号参数规范时，指定第 `N` 个参数需要所有前导参数，从第一个到第 `(N-1)` 个，都是指针。

所有形式的 `fscanf()` 函数都应允许检测输入字符串中依赖于语言的小数字符。小数字符在当前语言环境（类别 `LC_NUMERIC`）中定义。在 POSIX 语言环境中，或者在小数字符未定义的语言环境中，小数字符应默认为句点（`'.'`）。

应用程序应确保格式是一个字符字符串，如果有初始移位状态，则开始和结束于其初始移位状态，由零个或多个指令组成。每个指令由以下之一组成：一个

或多个空白字节；一个普通字符（既不是 `'%'` 也不是空白字节）；或一个转换说明。每个转换说明由字符 `'%'` [CX] 或字符序列 `"%n$"` 引入，之后按顺序出现以下内容：

- 一个可选的赋值抑制字符 `'*'`。
- 一个可选的非零十进制整数，指定最大字段宽度。
- [CX] 一个可选的赋值分配字符 `'m'`。
- 一个可选的长度修饰符，指定接收对象的大小。
- 一个指定要应用的转换类型的 `转换说明符` 字符。有效的转换说明符描述如下。

`fscanf()` 函数应依次执行格式的每个指令。当所有指令都已执行，或者如果指令失败（如下详述），函数应返回。失败被描述为输入失败（由于输入字节不可用）或匹配失败（由于输入不适当）。

由一个或多个空白字节组成的指令应通过读取输入直到第一个非空白字节来执行，该字节应保持未读状态，或者直到无法读取更多字节。该指令永不失败。

作为普通字符的指令应执行如下：从输入中读取下一个字节，并与组成指令的字节进行比较；如果比较显示它们不等效，指令应失败，不同的字节和后续字节应保持未读状态。类似地，如果文件结束、编码错误或读取错误阻止了字符被读取，指令应失败。

作为转换说明的指令定义了一组匹配的输入序列，如下所述对每个转换字符。转换说明应按以下步骤执行。

应跳过输入空白字节，除非转换说明包括 `[`、`c`、`C` 或 `n` 转换说明符。

应从输入中读取一个项，除非转换说明包括 `n` 转换说明符。输入项应定义为输入字节的最长序列（最多到任何指定的最大字段宽度，该宽度可能根据转换说明符以字符或字节测量），该序列是匹配序列的初始子序列。输入项之后的第一个字节（如果有）应保持未读状态。如果输入项的长度为 0，转换说明的执行应失败；此条件是匹配失败，除非文件结束、编码错误或读取错误阻止了从流中输入，在这种情况下是输入失败。

除了 `%` 转换说明符的情况下，输入项（或者在 `%n` 转换说明符的情况下，输入字节的计数）应被转换为适合转换字符的类型。如果输入项不是匹配序列，转换说明的执行失败；此条件是匹配失败。除非赋值被 `'*'` 抑制，转换的结果应放置在紧跟在 `format` 参数之后的第一个尚未接收转换结果的参数指向的对象中，如果转换说明由 `%` 引入，[CX] 或者如果在第 `n` 个参数中由字符序列 `"%n$"` 引入。如果此对象没有适当的类型，或者转换的结果无法在提供的空间中表示，行为是未定义的。

[CX] `c`、`s` 和 `[` 转换说明符应接受可选的赋值分配字符 `'m'`，这将导致分配内存缓冲区来保存转换结果。如果转换说明符是 `s` 或 `[`，分配的缓冲区应包括用于终止空字符（或宽字符）的空间。在这种情况下，对于转换说明的参数应是指针变量的引用，该变量将接收指向分配缓冲区的指针。系统应分配缓冲区，如同调用了 `malloc()` 一样。应用程序应负责在使用后释放内存。如果没有足够的内存分配缓冲区，函数应将 `errno` 设置为 `[ENOMEM]`，并导致转换错误。如果函数返回 `EOF`，任何通过使用赋值分配字符 `'m'` 为参数成功分配的内存在函数返回前应被释放。

长度修饰符

长度修饰符及其含义是：

- **hh**: 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `signed char` 或 `unsigned char` 的参数。
- **h**: 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `short` 或 `unsigned short` 的参数。
- **l** (小写字母 L): 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `long` 或 `unsigned long` 的参数；后续的 `a`、`A`、`e`、`E`、`f`、`F`、`g` 或 `G` 转换说明符应用于类型为指向 `double` 的参数；或者后续的 `c`、`s` 或 `[` 转换说明符应用于类型为指向 `wchar_t` 的参数。[CX] 如果指定了 `'m'` 赋值分配字符，转换应用于类型为指向指向 `wchar_t` 的指针的参数。
- **ll** (两个小写字母 L): 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `long long` 或 `unsigned long long` 的参数。
- **j**: 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `intmax_t` 或 `uintmax_t` 的参数。
- **z**: 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `size_t` 或相应有符号整数类型的参数。
- **t**: 指定后续的 `d`、`i`、`o`、`u`、`x`、`X` 或 `n` 转换说明符应用于类型为指向 `ptrdiff_t` 或相应 `unsigned` 类型的参数。
- **L**: 指定后续的 `a`、`A`、`e`、`E`、`f`、`F`、`g` 或 `G` 转换说明符应用于类型为指向 `long double` 的参数。

如果长度修饰符与除上述指定之外的任何转换说明符一起出现，行为是未定义的。

转换说明符

以下转换说明符是有效的：

- **d**: 匹配一个可选有符号的十进制整数，其格式与 `strtol()` 主题序列预期的相同，`base` 参数值为 10。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `int` 的指针。
- **i**: 匹配一个可选有符号的整数，其格式与 `strtol()` 主题序列预期的相同，`base` 参数值为 0。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `int` 的指针。
- **o**: 匹配一个可选有符号的八进制整数，其格式与 `strtoul()` 主题序列预期的相同，`base` 参数值为 8。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `unsigned` 的指针。
- **u**: 匹配一个可选有符号的十进制整数，其格式与 `strtoul()` 主题序列预期的相同，`base` 参数值为 10。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `unsigned` 的指针。
- **x**: 匹配一个可选有符号的十六进制整数，其格式与 `strtoul()` 主题序列预期的相同，`base` 参数值为 16。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `unsigned` 的指针。
- **a, e, f, g**: 匹配一个可选有符号的浮点数、无穷大或 NaN，其格式与 `strtod()` 主题序列预期的相同。在没有大小修饰符的情况下，应用程序应确保相应参数是指向 `float` 的指针。

如果 `fprintf()` 函数系列生成无穷大和 NaN（以浮点格式编码的符号实体）的字符串表示以支持 IEEE Std 754-1985，`fscanf()` 函数系列应将它们识别为输入。

- **s**: 匹配一个不是空白字节的字节序列。如果未指定 `'m'` 赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列和终止空字符代码，该代码应自动添加。[CX] 否则，应用程序应确保相应参数是指向指向 `char` 的指针。

如果存在 `l`（小写字母 L）限定符，输入是从初始移位状态开始的字符序列。每个字符应转换为宽字符，如同调用 `mbrtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。如果未指定 `'m'` 赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受序列和终止空宽字符，该字符应自动添加。[CX] 否则，应用程序应确保相应参数是指向指向 `wchar_t` 的指针。

- [: 匹配来自一组预期字节（扫描集）的非空字节序列。在这种情况下，应抑制对空白字节的正常跳过。如果未指定 '`m`' 赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列和终止空字节，该字节应自动添加。[CX] 否则，应用程序应确保相应参数是指向指向 `char` 的指针。

如果存在 `l`（小写字母 L）限定符，输入是从初始移位状态开始的字符序列。序列中的每个字符应转换为宽字符，如同调用 `mbrtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。如果未指定 '`m`' 赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受序列和终止空宽字符，该字符应自动添加。[CX] 否则，应用程序应确保相应参数是指向指向 `wchar_t` 的指针。

转换说明包括格式字符串中直到并包括匹配的右方括号 (`']'`) 的所有后续字节。方括号之间的字节（扫描列表）组成扫描集，除非左方括号之后的字节是插入符 (`'^'`)，在这种情况下，扫描集包含在插入符和右方括号之间的扫描列表中未出现的所有字节。如果转换说明以 `"[]"` 或 `"[^]"` 开始，右方括号包含在扫描列表中，下一个右方括号是结束转换说明的匹配右方括号；否则，第一个右方括号是结束转换说明的那个。如果 `'-'` 在扫描列表中并且不是第一个字符，也不是第一个字符为 `'^'` 时的第二个字符，也不是最后一个字符，行为是实现定义的。

- `c`: 匹配由字段宽度指定的数量的字节序列（如果转换说明中没有字段宽度则为 1）。不添加空字节。在这种情况下，应抑制对空白字节的正常跳过。如果未指定 '`m`' 赋值分配字符，应用程序应确保相应参数是指向 `char`、`signed char` 或 `unsigned char` 数组初始字节的指针，该数组足够大以接受序列。[CX] 否则，应用程序应确保相应参数是指向指向 `char` 的指针。

如果存在 `l`（小写字母 L）限定符，输入应是从初始移位状态开始的字符序列。序列中的每个字符转换为宽字符，如同调用 `mbrtowc()` 函数一样，转换状态由在第一个字符转换前初始化为零的 `mbstate_t` 对象描述。不添加空宽字符。如果未指定 '`m`' 赋值分配字符，应用程序应确保相应参数是指向 `wchar_t` 数组的指针，该数组足够大以接受生成的宽字符序列。[CX] 否则，应用程序应确保相应参数是指向指向 `wchar_t` 的指针。

- `p`: 匹配实现定义的序列集合，该集合应与相应 `fprintf()` 函数的 `%p` 转换说明符生成的序列集合相同。应用程序应确保相应参数是指向指向 `void` 的指针。输入项的解释是实现定义的。如果输入项是同一程序执行期间先前转换的值，结果指针应与该值比较相等；否则，`%p` 转换说明符的行为是未定义的。
- `n`: 不消耗输入。应用程序应确保相应参数是指向整数的指针，该整数将被写入迄今为止通过此对 `fscanf()` 函数的调用从输入读取的字节数。`%n`

转换说明符的执行不应增加函数执行完成时返回的赋值计数。不应转换参数，但应消耗一个。如果转换说明包括赋值抑制字符或字段宽度，行为是未定义的。

- **C**: [XSI] 等效于 `lc`。
- **S**: [XSI] 等效于 `ls`。
- **%**: 匹配单个 `'%'` 字符；不发生转换或赋值。完整的转换说明应为 `%%`。

如果转换说明无效，行为是未定义的。

转换说明符 `A`、`E`、`F`、`G` 和 `X` 也是有效的，并应分别等效于 `a`、`e`、`f`、`g` 和 `x`。

执行规则

如果在输入期间遇到文件结束，转换应终止。如果在读取匹配当前转换说明（除了 `%n`）的任何字节之前发生文件结束（除了允许的前导空白字节），当前转换说明的执行应以输入失败终止。否则，除非当前转换说明的执行以匹配失败终止，后续转换说明（如果有）的执行应以输入失败终止。

在 `sscanf()` 中到达字符串末尾应等效于在 `fscanf()` 中遇到文件结束。

如果转换在冲突输入上终止，有问题的输入在输入中保持未读状态。任何尾随空白字节（包括换行字符）应保持未读状态，除非被转换说明匹配。字面量匹配和抑制赋值的成功只能通过 `%n` 转换说明直接确定。

[CX] `fscanf()` 和 `scanf()` 函数可以标记与 `stream` 关联的文件的最后数据访问时间戳以供更新。第一次成功执行使用 `stream` 的 `fgetc()`、`fgets()`、 `fread()`、`getc()`、`getchar()`、`getdelim()`、`getline()`、`fscanf()` 或 `scanf()` 并返回不是由先前 `ungetc()` 调用提供的数据时，应标记最后数据访问时间戳以供更新。

返回值

成功完成后，这些函数应返回成功匹配和赋值的输入项的数量；在早期匹配失败的情况下，此数字可能为零。如果输入在第一次转换（如果有）完成之前结束，并且没有发生匹配失败，应返回 EOF。如果在第一次转换（如果有）完成之前发生错误，并且没有发生匹配失败，应返回 EOF [CX] 并设置 `errno` 以指示错误。如果发生错误，应设置流的错误指示器。

错误

关于 `fscanf()` 函数失败和可能失败的条件，请参考 `fgetc()` 或 `fgetwc()`。

此外，`fscanf()` 函数应在以下情况下失败：

- **[EILSEQ] [CX]** 输入字节序列不形成有效字符。
- **[ENOMEM]** 存储空间不足。

此外，`fscanf()` 函数可能在以下情况下失败：

- **[EINVAL] [CX]** 参数不足。

示例

基本示例

调用：

```
int i, n; float x; char name[50];
n = scanf("%d%f%s", &i, &x, name);
```

使用输入行：

```
25 54.32E-1 Hamster
```

将值 3 赋给 `n`，值 25 赋给 `i`，值 5.432 赋给 `x`，`name` 包含字符串 `"Hamster"`。

高级示例

调用：

```
int i; float x; char name[50];
(void) scanf("%2d%f%d %[0123456789]", &i, &x, name);
```

使用输入：

```
56789 0123 56a72
```

将 56 赋给 `i`，789.0 赋给 `x`，跳过 0123，并将字符串 `"56\0"` 放入 `name`。下一次对 `getchar()` 的调用应返回字符 `'a'`。

将数据读入数组

以下调用使用 `fscanf()` 从标准输入读取三个浮点数到 `input` 数组中。

```
float input[3];
fscanf(stdin, "%f %f %f", input, input+1, input+2);
```

应用程序使用

如果调用 `fscanf()` 的应用程序具有任何类型为 `wint_t` 或 `wchar_t` 的对象，它还必须包含 `<wchar.h>` 头文件以定义这些对象。

对于如同通过 `malloc()` 一样分配内存的函数，应用程序应在不再需要时通过调用 `free()` 释放此类内存。对于 `fscanf()`，这是通过使用 `'m'` 赋值分配字符分配的内存。

基本原理

扫描集中允许的字符集限于单字节字符。在其他类似地方，允许多字节字符，但为了与 ISO C 标准保持一致，这里没有这样做。需要此功能的应用程序可以使用相应的宽字符函数来实现所需结果。

未来方向

无。

另请参见

- `2.5 标准 I/O 流`
- `fprintf()`
- `getc()`
- `setlocale()`
- `strtod()`
- `strtol()`
- `strtoul()`

- `wcrtomb()`

XBD 引用

- `7. 语言环境`
- `<inttypes.h>`
- `<langinfo.h>`
- `<stdio.h>`
- `<wchar.h>`

变更历史

第一次发布于 Issue 1。

源自 SVID 的 Issue 1。

Issue 5

与 ISO/IEC 9899:1990/Amendment 1:1995 (E) 对齐。具体来说，`l` (小写字母 L) 限定符现在为 `c`、`s` 和 `[` 转换说明符定义。

DESCRIPTION 更新以指示如果 `fprintf()` 函数系列可以生成无穷大和 NaN，那么它们被 `fscanf()` 系列识别。

Issue 6

应用了 Open Group Corrigenda U021/7 和 U028/10。这些更正了文本中多次出现的 "characters"，已被术语 "bytes" 替换。

规范性文本更新以避免对应用程序要求使用术语 "must"。

为与 ISO/IEC 9899:1999 标准对齐进行以下更改：

- 更新了 `fscanf()`、`scanf()` 和 `sscanf()` 的原型。
- 更新了 DESCRIPTION。
- 添加了 `hh`、`ll`、`j`、`t` 和 `z` 长度修饰符。
- 添加了 `a`、`A` 和 `F` 转换字符。

DESCRIPTION 更新以一致地使用术语 "转换说明符" 和 "转换说明"。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #170。

应用了 SD5-XSH-ERN-9，将 DESCRIPTION 中的 `fscanf()` 更正为 `scanf()`。

应用了 SD5-XSH-ERN-132，添加了赋值分配字符 '`m`'。

与 `%n$` 形式转换说明相关的功能从 XSI 选项移至 Base。

进行了与支持细粒度时间戳相关的更改。

APPLICATION USAGE 部分更新以阐明内存如同通过 `malloc()` 分配。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0185 [302], XSH/TC1-2008/0186 [90], 和 XSH/TC1-2008/0187 [14]。XSH/TC1-2008/0186 [90] 更改了 RETURN VALUE 部分的第二句话，以与 ISO C 标准下一版本预期的措辞变更对齐。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0135 [936]。

Issue 8

应用了 Austin Group Defect 1163，阐明格式字符串中空白字符的处理。

应用了 Austin Group Defect 1173，阐明赋值分配字符 '`m`' 的描述。

应用了 Austin Group Defect 1302，将这些函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group Defect 1330，移除了过时的接口。

应用了 Austin Group Defect 1375，将 "终止空字符" 更改为 "终止空字符（或宽字符）"。

应用了 Austin Group Defect 1562，阐明确保格式是字符字符串，开始和结束于其初始移位状态（如果有）是应用程序的责任。

应用了 Austin Group Defect 1624，更改了 RETURN VALUE 部分。

1.226. strcat

SYNOPSIS

```
#include <string.h>

char *strcat(char *restrict s1, const char *restrict s2);
```

DESCRIPTION

`strcat()` 函数应将 `s2` 所指向的字符串的副本（包括终止的 NUL 字符）追加到 `s1` 所指向的字符串的末尾。`s2` 的初始字节覆盖 `s1` 末尾的 NUL 字符。如果在重叠的对象之间进行复制，则行为是未定义的。

`strcat()` 函数在有效输入时不应改变 `errno` 的设置。

RETURN VALUE

`strcat()` 函数应返回 `s1`；没有保留返回值来指示错误。

ERRORS

未定义错误。

以下部分为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

此版本与 ISO C 标准对齐；这不会影响与 XPG3 应用程序的兼容性。此函数的可靠错误检测从未得到保证。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `strncat()`
- XBD `<string.h>`

CHANGE HISTORY

首次在 Issue 1 中发布。派生自 SVID 的 Issue 1。

Issue 6

`strcat()` 原型已更新，以与 ISO/IEC 9899:1999 标准对齐。

Issue 8

应用了 Austin Group Defect 448，增加了 `strcat()` 在有效输入时不改变 `errno` 设置的要求。

1.227. strchr

SYNOPSIS (概要)

```
#include <string.h>

char *strchr(const char *s, int c);
```

DESCRIPTION (描述)

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`strchr()` 函数应在 `s` 指向的字符串中定位 `c` (转换为 `char` 类型) 第一次出现的位置。终止的 NUL 字符被认为是字符串的一部分。

[CX] `strchr()` 函数在有效输入时不应改变 `errno` 的设置。

RETURN VALUE (返回值)

完成后, `strchr()` 应返回指向该字节的指针, 如果未找到该字节则返回空指针。

ERRORS (错误)

未定义错误。

以下部分为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `strrchr()`
- XBD `<string.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 8

应用了 Austin Group 缺陷 448，增加了 `strchr()` 在有效输入时不改变 `errno` 设置的要求。

补充信息结束。

1.228. strcmp – 比较两个字符串

概要 (SYNOPSIS)

```
#include <string.h>

int strcmp(const char *s1, const char *s2);
```

描述 (DESCRIPTION)

本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`strcmp()` 函数应将 `s1` 指向的字符串与 `s2` 指向的字符串进行比较。

非零返回值的符号应由被比较字符串中第一对相异字节（两者均解释为 `unsigned char` 类型）的值的差值符号确定。

`strcmp()` 函数在有效输入时不应改变 `errno` 的设置。

返回值 (RETURN VALUE)

完成时，如果 `s1` 指向的字符串分别大于、等于或小于 `s2` 指向的字符串，
`strcmp()` 应返回一个大于、等于或小于 0 的整数。

错误 (ERRORS)

未定义错误。

示例 (EXAMPLES)

检查密码条目 (Checking a Password Entry)

以下示例将从标准输入读取的信息与用户条目的名称值进行比较。如果 `strcmp()` 函数返回 0（表示匹配），将进一步检查用户是否输入了正确的旧密码。`crypt()` 函数应使用 `passwd` 结构中加密密码的值作为盐值来加密用户

输入的旧密码。如果此值与结构中加密的 `passwd` 的值匹配，则输入的密码 `oldpasswd` 是正确的用户密码。最后，程序加密新密码，以便将信息存储在 `passwd` 结构中。

```
#include <string.h>
#include <unistd.h>
#include <stdio.h>
...
int valid_change;
struct passwd *p;
char user[100];
char oldpasswd[100];
char newpasswd[100];
char savepasswd[100];
...
if (strcmp(p->pw_name, user) == 0) {
    if (strcmp(p->pw_passwd, crypt(oldpasswd, p->pw_passwd)) ==
        strcpy(savepasswd, crypt(newpasswd, user)));
    p->pw_passwd = savepasswd;
    valid_change = 1;
}
else {
    fprintf(stderr, "Old password is not valid\n");
}
...
...
```

应用用法 (APPLICATION USAGE)

无。

基本原理 (RATIONALE)

无。

未来方向 (FUTURE DIRECTIONS)

无。

另请参见 (SEE ALSO)

- `strncmp()`
- `<string.h>`

变更历史 (CHANGE HISTORY)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 8

应用了 Austin Group 缺陷 448，增加了 `strcmp()` 在有效输入时不改变 `errno` 设置的要求。

1.229. strcoll

SYNOPSIS

```
#include <string.h>

int strcoll(const char *s1, const char *s2);

int strcoll_l(const char *s1, const char *s2, locale_t locale);
```

DESCRIPTION

对于 `strcoll()`：本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 此卷遵循 ISO C 标准。

`strcoll()` 和 `strcoll_l()` 函数应将 `s1` 指向的字符串与 `s2` 指向的字符串进行比较，两个字符串分别根据当前 `locale` 的 `LC_COLLATE` 类别或 `locale` 表示的 `locale` 进行适当的解释。

如果成功，`strcoll()` 和 `strcoll_l()` 函数不应更改 `errno` 的设置。

由于没有保留返回值来指示错误，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strcoll()` 或 `strcoll_l()`，然后检查 `errno`。

如果 `strcoll_l()` 的 `locale` 参数是特殊的 `locale` 对象 `LC_GLOBAL_LOCALE` 或不是有效的 `locale` 对象句柄，则行为未定义。

RETURN VALUE

成功完成后，`strcoll()` 应返回一个大于、等于或小于 0 的整数，具体取决于当 `s1` 指向的字符串在根据当前 `locale` 进行适当解释时，是大于、等于还是小于 `s2` 指向的字符串。出错时，`strcoll()` 可以设置 `errno`，但没有保留返回值来指示错误。

成功完成后，`strcoll_l()` 应返回一个大于、等于或小于 0 的整数，具体取决于当 `s1` 指向的字符串在根据 `locale` 表示的 `locale` 进行适当解释时，是大于、等于还是小于 `s2` 指向的字符串。出错时，`strcoll_l()` 可以设置 `errno`，但没有保留返回值来指示错误。

ERRORS

这些函数可能在以下情况下失败：

- **[EINVAL]**
- **s1** 或 **s2** 参数包含排序序列域之外的字符。

EXAMPLES

比较节点

以下示例使用应用程序定义的函数 `node_compare()` 来基于 `string` 字段的字母顺序比较两个节点。

```
#include <string.h>
...
struct node { /* 这些存储在表中。 */
    char *string;
    int length;
};
...
int node_compare(const void *node1, const void *node2)
{
    return strcoll(((const struct node *)node1)->string,
                   ((const struct node *)node2)->string);
}
...
```

APPLICATION USAGE

`strxfrm()` 和 `strcmp()` 函数应该用于对大型列表进行排序。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `alphasort()`
- `strcmp()`
- `strxfrm()`

XBD `<string.h>`

CHANGE HISTORY

首次发布于 Issue 3。

Issue 5

更新 DESCRIPTION 以指示如果函数成功则 `errno` 不会更改。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 添加了 [EINVAL] 可选错误条件。

添加了示例。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `strcoll_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0593 [283] 和 XSH/TC1-2008/0594 [283]。

1.230. strcpy

概要

```
#include <string.h>

[CX] char *stpcpy(char *restrict s1, const char *restrict s2);
char *strcpy(char *restrict s1, const char *restrict s2);
```

描述

对于 `strcpy()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 卷服从 ISO C 标准。

[CX] `stpcpy()` 和 `strcpy()` 函数应将由 `s2` 指向的字符串（包括终止的 NUL 字符）复制到由 `s1` 指向的数组中。

如果在重叠的对象之间进行复制，则行为未定义。

[CX] `strcpy()` 和 `stpcpy()` 函数在有效输入时不应更改 `errno` 的设置。

返回值

[CX] `stpcpy()` 函数应返回指向复制到 `s1` 缓冲区中的终止 NUL 字符的指针。

`strcpy()` 函数应返回 `s1`。

没有保留任何返回值来指示错误。

错误

未定义任何错误。

以下章节为参考信息。

示例

在单个缓冲区中构造多部分消息

```
#include <string.h>
#include <stdio.h>

int
main (void)
{
    char buffer[10];
    char *name = buffer;

    name = stpcpy (stpcpy (stpcpy (name, "ice"), "-"), "cream");
    puts (buffer);
    return 0;
}
```

初始化字符串

以下示例将字符串 "-----" 复制到 `permstring` 变量中。

```
#include <string.h>
...
static char permstring[11];
...
strcpy(permstring, "-----");
...
```

存储键和数据

以下示例使用 `malloc()` 为键分配空间，然后使用 `strcpy()` 将键放置在那里。然后使用 `malloc()` 为数据分配空间，并使用 `strcpy()` 将数据放置在那里。（用户定义的函数 `dbfree()` 释放先前分配给 `struct element *` 类型数组的内存。）

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
...
/* 用于读取数据并存储的结构体。 */
struct element {
```

```
char *key;
char *data;
};

struct element *tbl, *curtbl;
char *key, *data;
int count;
...
void dbfree(struct element *, int);
...
if ((curtbl->key = malloc(strlen(key) + 1)) == NULL) {
    perror("malloc");
    dbfree(tbl, count);
    return NULL;
}
strcpy(curtbl->key, key);

if ((curtbl->data = malloc(strlen(data) + 1)) == NULL) {
    perror("malloc");
    free(curtbl->key);
    dbfree(tbl, count);
}
strcpy(curtbl->data, data);
...
```

应用程序使用

字符移动在不同的实现中执行方式不同。因此，重叠移动可能会产生意外结果。

此版本与 ISO C 标准保持一致；这不会影响与 XPG3 应用程序的兼容性。此函数的可靠错误检测从未得到保证。

原理

无。

未来方向

无。

另请参阅

- [strncpy\(\)](#)

- `wcscpy()`
- XBD `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

`strcpy()` 原型已更新，以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 中添加了 `stpcpy()` 函数。

Issue 8

应用了 Austin Group Defect 448，增加了 `strcpy()` 和 `stpcpy()` 在有效输入时不更改 `errno` 设置的要求。

应用了 Austin Group Defect 1787，更改了 NAME 部分。

1.231. strcspn

概要

```
#include <string.h>

size_t strcspn(const char *s1, const char *s2);
```

描述

`strcspn()` 函数应该计算由 `s1` 指向的字符串的最大初始段长度（以字节为单位），该段完全由不来自 `s2` 指向的字符串的字节组成。

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

在有效输入上，`strcspn()` 函数不应改变 `errno` 的设置。

返回值

`strcspn()` 函数应返回计算的 `s1` 指向字符串段的长度；没有保留返回值来表示错误。

错误

未定义任何错误。

以下各节为参考信息。

示例

无。

应用程序用法

无。

原理

无。

未来方向

无。

参见

- `strspn()`
- XBD `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

RETURN VALUE 部分更新为说明 `strcspn()` 返回 `s1` 的长度，而非之前所述的 `s1` 本身。

Issue 6

应用了 The Open Group Corrigendum U030/1。RETURN VALUE 部分的文本更新为说明返回的是计算的段长度，而非 `s1` 的长度。

Issue 8

应用了 Austin Group Defect 448，添加了要求：在有效输入上，`strcspn()` 不改变 `errno` 的设置。

1.232. strerror, strerror_l, strerror_r — 获取错误消息字符串

SYNOPSIS (概要)

```
#include <string.h>

char *strerror(int errnum);

/* XSI 扩展 */
char *strerror_l(int errnum, locale_t locale);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

DESCRIPTION (描述)

对于 strerror():

- 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 标准遵循 ISO C 标准。

strerror() 函数应将 errnum 中的错误编号映射为依赖于语言环境的错误消息字符串，并返回指向该字符串的指针。通常，errnum 的值来自 errno，但 strerror() 应将任何 int 类型的值映射为消息。

应用程序不应修改返回的字符串。后续的 strerror() 调用，或同一线程中后续的 strerror_l() 调用，可能会使返回的字符串指针失效或覆盖字符串内容。如果调用线程终止，返回的指针和字符串内容也可能失效。

该字符串可能被同一线程中后续的 strerror_l() 调用覆盖。

strerror() 返回的错误消息字符串的内容应由当前语言环境中 LC_MESSAGES 类别的设置决定。

实现的行为应如同 POSIX.1-2024 标准卷中定义的任何函数都不调用 strerror()。

如果成功，strerror() 和 strerror_l() 函数不应更改 errno 的设置。

由于没有保留返回值来指示 strerror() 的错误，希望检查错误情况的应用程序应将 errno 设置为 0，然后调用 strerror()，再检查 errno。类似地，由于 strerror_l() 需要为某些错误返回字符串，希望检查所有错误情况的应用程序应将 errno 设置为 0，然后调用 strerror_l()，再检查 errno。

strerror() 函数不需要是线程安全的；但是，strerror() 应避免与所有其他函数发生数据竞争。

strerror_l() 函数应将 errnum 中的错误编号映射为由 locale 表示的语言环境中的依赖于语言环境的错误消息字符串，并返回指向该字符串的指针。

strerror_r() 函数应将 errnum 中的错误编号映射为依赖于语言环境的错误消息字符串，并应在 strerrbuf 指向的缓冲区（长度为 buflen）中返回该字符串。

如果 errnum 的值是有效的错误编号，消息字符串应指示发生了什么错误；如果 errnum 的值是零，消息字符串应为空字符串或指示没有发生错误；否则，如果这些函数成功完成，消息字符串应指示发生了未知错误。

如果传递给 strerror_l() 的 locale 参数是特殊的语言环境对象 LC_GLOBAL_LOCALE 或不是有效的语言环境对象句柄，则行为未定义。

RETURN VALUE (返回值)

完成时，无论成功与否，strerror() 都应返回指向生成的消息字符串的指针。错误时可能设置 errno，但没有保留返回值来指示错误。

成功完成时，strerror_l() 应返回指向生成的消息字符串的指针。如果 errnum 不是有效的错误编号，可能将 errno 设置为 [EINVAL]，但仍应返回指向消息字符串的指针。如果发生任何其他错误，应设置 errno 来指示错误，并返回空指针。

成功完成时，strerror_r() 应返回 0。否则，应返回错误编号来指示错误。

ERRORS (错误)

这些函数可能在以下情况下失败：

- **EINVAL** - errnum 的值既不是有效的错误编号也不是零。

strerror_r() 函数应在以下情况下失败：

- **ERANGE** - 通过 strerrbuf 和 buflen 提供的存储空间不足以包含生成的消息字符串。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

历史上在某些实现中，调用 perror() 会覆盖 strerror() 返回的指针指向的字符串。这样的实现不符合 ISO C 标准；但是，如果应用程序开发者希望其应用程序可移植到此类实现，应注意这种行为。

应用程序应使用 strerror_l() 而不是 strerror() 或 strerror_r()，以避免线程安全问题和某些实现中这些函数可能的替代（非符合）版本。

RATIONALE (基本原理)

strerror_l() 函数需要是线程安全的，从而消除了对等效于 strerror_r() 函数的需求。

本标准的早期版本没有明确要求 strerror() 和 strerror_r() 返回的错误消息字符串提供有关错误的任何信息。本版本的标准要求任何成功完成都要提供有意义的消息。

由于没有保留返回值来指示 strerror() 错误，但所有调用（无论成功与否）都必须返回指向消息字符串的指针，错误时 strerror() 可以返回指向空字符串的指针或指向可以打印的有意义字符串的指针。

请注意，[EINVAL] 错误条件是“可能失败”的错误。如果提供了无效的错误编号作为 errnum 的值，应用程序应准备好处理以下任何情况：

- 1. 错误（没有有意义的消息）：**errno 设置为 [EINVAL]，返回值是指向空字符串的指针。
- 2. 成功完成：**errno 未更改，返回值指向类似 "unknown error" 或 "error number xxx"（其中 xxx 是 errnum 的值）的字符串。
- 3. #1 和 #2 的组合：**errno 设置为 [EINVAL]，返回值指向类似 "unknown error" 或 "error number xxx"（其中 xxx 是 errnum 的值）的字符串。由于应用程序经常使用 strerror() 的返回值作为 fprintf() 等函数的参数（而不检查返回值），并且应用程序无法解析错误消息字符串来确定 errnum 是否表示有效的错误编号，鼓励实现采用 #3。类似地，当 errnum 的值不是有效的错误编号时，鼓励 strerror_r() 返回 [EINVAL] 并在 strerrbuf 指向的缓冲区中放入类似 "unknown error" 或 "error number xxx" 的字符串。

此外，对于除零外的任何大小，鼓励在因 [ERANGE] 失败时对 strerrbuf 进行空终止。

一些应用程序依赖于能够在调用没有保留值来指示错误的函数之前将 errno 设置为 0，然后调用 strerror(errno) 来检测是否发生错误（因为 errno 更改）或指

示成功（因为 `errno` 保持为零）。这种使用模式要求 `strerror(0)` 成功并提供有用的结果。标准的前一个版本没有指定当 `errnum` 为零时的行为。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- [perror\(\)](#)
-

CHANGE HISTORY (变更历史)

首次发布于 Issue 3。

Issue 5

DESCRIPTION 更新为指示如果函数成功则不更改 `errno`。

在 DESCRIPTION 中添加了说明 `strerror()` 函数不需要可重入的注释。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 部分，添加了可能设置 `errno` 的事实。
- 添加了 [EINVAL] 可选错误条件。

规范性文本更新为避免对应用程序要求使用"必须"一词。

根据 IEEE PASC Interpretation 1003.1c #39 添加了 `strerror_r()` 函数。

`strerror_r()` 函数被标记为线程安全函数选项的一部分。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #072，更新了 ERRORS 部分。

应用了 Austin Group Interpretation 1003.1-2001 #156。

应用了 Austin Group Interpretation 1003.1-2001 #187，阐明了当生成的错误消息为空字符串时的行为。

应用了 SD5-XSH-ERN-191，更新了 APPLICATION USAGE 部分。

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 strerror_l() 函数。

strerror_r() 函数从线程安全函数选项移至 Base。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0595 [75], XSH/TC1-2008/0596 [447], XSH/TC1-2008/0597 [382,428], XSH/TC1-2008/0598 [283], XSH/TC1-2008/0599 [382,428], XSH/TC1-2008/0600 [283]，和 XSH/TC1-2008/0601 [382,428]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0339 [656]。

Issue 8

应用了 Austin Group Defect 398，将 [ERANGE] 错误从"可能失败"更改为"应失败"。

应用了 Austin Group Defect 655，更改了 APPLICATION USAGE 部分。

应用了 Austin Group Defect 1302，使 strerror() 函数与 ISO/IEC 9899:2018 标准对齐。

1.233. strerror, strerror_l, strerror_r — 获取错误消息字符串

概要 (SYNOPSIS)

```
#include <string.h>

char *strerror(int errnum);

[CX] char *strerror_l(int errnum, locale_t locale);
int strerror_r(int errnum, char *strerrbuf, size_t buflen);
```

描述 (DESCRIPTION)

对于 `strerror()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

`strerror()` 函数应将 `errnum` 中的错误号映射到依赖于区域设置的错误消息字符串，并应返回指向该字符串的指针。通常，`errnum` 的值来自 `errno`，但 `strerror()` 应将任何 `int` 类型的值映射到消息。

应用程序不应修改返回的字符串。[CX] 返回的字符串指针可能被后续对 `strerror()` 的调用所无效化，或者字符串内容可能被覆盖，[CX] 或者在同一线程中后续对 `strerror_l()` 的调用所覆盖。如果调用线程终止，返回的指针和字符串内容也可能被无效化。

[CX] 字符串可能被同一线程中后续对 `strerror_l()` 的调用所覆盖。

`strerror()` 返回的错误消息字符串的内容应由当前区域设置中的 `LC_MESSAGES` 类别设置决定。

实现的行为应如同本 POSIX.1-2024 卷中定义的任何函数都不调用 `strerror()`。

[CX] 如果成功，`strerror()` 和 `strerror_l()` 函数不应更改 `errno` 的设置。

由于没有保留返回值来指示 `strerror()` 的错误，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strerror()`，然后检查 `errno`。类似地，由于 `strerror_l()` 被要求为某些错误返回字符串，希望检查所有错误情

况的应用程序应将 `errno` 设置为 0，然后调用 `strerror_l()`，然后检查 `errno`。

`strerror()` 函数不需要是线程安全的；但是，`strerror()` 应避免与所有其他函数发生数据竞争。

[CX] `strerror_l()` 函数应将 `errnum` 中的错误号映射到由 `locale` 表示的区域设置中依赖于区域设置的错误消息字符串，并应返回指向该字符串的指针。

`strerror_r()` 函数应将 `errnum` 中的错误号映射到依赖于区域设置的错误消息字符串，并应将字符串返回到由 `strerrbuf` 指向的缓冲区中，长度为 `buflen`。

[CX] 如果 `errnum` 的值是有效的错误号，消息字符串应指示发生了什么错误；如果 `errnum` 的值为零，消息字符串应为空字符串或指示没有发生错误；否则，如果这些函数成功完成，消息字符串应指示发生了未知错误。

[CX] 如果 `strerror_l()` 的 `locale` 参数是特殊的区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

返回值 (RETURN VALUE)

完成时，无论成功与否，`strerror()` 都应返回指向生成的消息字符串的指针。[CX] 出错时 `errno` 可能被设置，但没有保留返回值来指示错误。

成功完成时，`strerror_l()` 应返回指向生成的消息字符串的指针。如果 `errnum` 不是有效的错误号，`errno` 可能被设置为 `[EINVAL]`，但仍应返回指向消息字符串的指针。如果发生任何其他错误，`errno` 应被设置为指示错误，并返回空指针。

成功完成时，`strerror_r()` 应返回 0。否则，应返回错误号以指示错误。

错误 (ERRORS)

这些函数可能在以下情况下失败：

- **[EINVAL]** [CX] `errnum` 的值既不是有效的错误号也不是零。

`strerror_r()` 函数应在以下情况下失败：

- **[ERANGE]** [CX] 通过 `strerrbuf` 和 `buflen` 提供的存储空间不足以包含生成的消息字符串。

示例 (EXAMPLES)

无。

应用程序用法 (APPLICATION USAGE)

历史上，在某些实现中，对 `perror()` 的调用会覆盖 `strerror()` 返回的指针所指向的字符串。这样的实现不符合 ISO C 标准；然而，如果应用程序开发人员希望他们的应用程序可以移植到这样的实现，应该注意这种行为。

应用程序应使用 `strerror_l()` 而不是 `strerror()` 或 `strerror_r()`，以避免线程安全问题以及在某些实现中这些函数的替代（非符合）版本。

基本原理 (RATIONALE)

要求 `strerror_l()` 函数是线程安全的，从而消除了对等同于 `strerror_r()` 函数的需求。

本标准的早期版本没有明确要求 `strerror()` 和 `strerror_r()` 返回的错误消息字符串提供有关错误的任何信息。本版本的标准要求任何成功完成时都要提供有意义的消息。

由于没有保留返回值来指示 `strerror()` 错误，但所有调用（无论成功与否）都必须返回指向消息字符串的指针，出错时 `strerror()` 可以返回指向空字符串的指针或返回指向可打印的有意义字符串的指针。

请注意，`[EINVAL]` 错误条件是一个可能失败（may fail）的错误。如果提供了无效的错误号作为 `errnum` 的值，应用程序应准备处理以下任何情况：

1. 错误（没有有意义的消息）：`errno` 被设置为 `[EINVAL]`，返回值是指向空字符串的指针。
2. 成功完成：`errno` 未更改，返回值指向类似 "unknown error" 或 "error number xxx"（其中 `xxx` 是 `errnum` 的值）的字符串。

3. **1.233.1. 1 和 #2 的组合：`errno` 被设置为 `[EINVAL]`，返回值指向类似 "unknown error" 或 "error number xxx"**

`number xxx"` (其中`xxx`是`errnum`的值) 的字符串。由于应用程序经常使用`strerror()`的返回值作为`fprintf()`等函数的参数 (而不检查返回值), 并且应用程序无法解析错误消息字符串来确定`errnum`是否代表有效的错误号, 鼓励实现实现 #3。类似地, 鼓励当`errnum`的值不是有效的错误号时, 让`strerror_r()`返回`[EINVAL]` 并在`strerrbuf`指向的缓冲区中放入类似 "`unknown error`" 或 "`error number xxx`" 的字符串。

此外, 鼓励在因 `[ERANGE]` 失败时 (对于 `buflen` 非零的任何大小) 将 `strerrbuf` 以 `null` 终止。

一些应用程序依赖于能够在调用没有保留值来指示错误的函数之前将 `errno` 设置为 0, 然后调用 `strerror(errno)` 来检测是否发生了错误 (因为 `errno` 更改了) 或指示成功 (因为 `errno` 保持为零)。这种使用模式要求 `strerror(0)` 成功并产生有用的结果。标准的早期版本没有指定当 `errnum` 为零时的行为。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- `perror()`
- XBD `<string.h>`

变更历史 (CHANGE HISTORY)

首次在 Issue 3 中发布。

Issue 5

DESCRIPTION 已更新，以指示如果函数成功则不更改 `errno`。

在 DESCRIPTION 中添加了一条注释，指出 `strerror()` 函数不需要是可重入的。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 部分，添加了 `errno` 可能被设置的事实。
- 添加了 [EINVAL] 可选错误条件。

规范性文本已更新，以避免对应用程序要求使用 "must" 一词。

为响应 IEEE PASC Interpretation 1003.1c #39，添加了 `strerror_r()` 函数。

`strerror_r()` 函数被标记为线程安全函数选项的一部分。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #072，更新了 ERRORS 部分。

应用了 Austin Group Interpretation 1003.1-2001 #156。

应用了 Austin Group Interpretation 1003.1-2001 #187，阐明了当生成的错误消息为空字符串时的行为。

应用了 SD5-XSH-ERN-191，更新了 APPLICATION USAGE 部分。

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 添加了 `strerror_l()` 函数。

`strerror_r()` 函数从线程安全函数选项移动到 Base。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0595 [75], XSH/TC1-2008/0596 [447], XSH/TC1-2008/0597 [382,428], XSH/TC1-2008/0598 [283], XSH/TC1-2008/0599 [382,428], XSH/TC1-2008/0600 [283], 和 XSH/TC1-2008/0601 [382,428]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0339 [656]。

Issue 8

应用了 Austin Group Defect 398，将 [ERANGE] 错误从 "可能失败" (may fail) 更改为 "应失败" (shall fail)。

应用了 Austin Group Defect 655，更改了 APPLICATION USAGE 部分。

应用了 Austin Group Defect 1302，使 `strerror()` 函数与 ISO/IEC 9899:2018 标准对齐。

1.234. strftime, strftime_l — 将日期和时间转换为字符串

概要

```
#include <time.h>

size_t strftime(char *restrict s, size_t maxsize,
                const char *restrict format, const struct tm *re
[CX] size_t strftime_l(char *restrict s, size_t maxsize,
                      const char *restrict format, const struct
locale_t locale);
```

描述

对于 `strftime()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`strftime()` 函数应按照 `format` 指向的字符串控制，将字节放入 `s` 指向的数组中。应用程序应确保格式是字符串，如果有移位状态，则应以其初始移位状态开始和结束。格式字符串由零个或多个转换规范和普通字符组成。

每个转换规范由 '%' 字符引入，其后按顺序出现以下内容：

- [CX] 可选标志：
 - `0` - 零字符 ('0')，指定用作填充字符的是 '0'
 - `+` - 加号字符 ('+')，指定用作填充字符的是 '0'，并且当且仅当生成的字段占用超过四个字节来表示年份（对于 %F、%G 或 %Y）或超过两个字节来表示年份除以 100（对于 %C）时，如果正在处理的年份大于等于零则包含前导加号字符，如果年份小于零则包含前导连字符减号 ('-')。
 - 默认填充字符未指定。
- 可选的最小字段宽度。如果转换后的值（包括任何前导 '+' 或 '-' 符号）的字节数少于最小字段宽度且填充字符不是 NUL 字符，则输出应在左侧（在任何前导 '+' 或 '-' 符号之后）用填充字符填充。

- 可选的 E 或 O 修饰符。
- 终止转换说明符字符，指示要应用的转换类型。

[CX] 如果指定了多个标志字符、指定了标志字符但没有最小字段宽度、指定了最小字段宽度但没有标志字符、将修饰符与标志或最小字段宽度一起指定，或者为除 C、F、G 或 Y 之外的任何转换说明符指定了最小字段宽度，则结果未指定。

所有普通字符（包括终止的 NUL 字符）都原样复制到数组中。如果在重叠的对象之间进行复制，则行为未定义。数组中放置的字节数不超过 `maxsize`。每个转换说明符将被适当的字符替换，如下列表所述。适当的字符使用当前语言环境的 `LC_TIME` 类别和 `timeptr` 指向的分解时间结构的零个或多个成员的值确定，如描述中的括号所示。如果任何指定值超出正常范围，则存储的字符未指定。

[CX] `strftime_l()` 函数应等同于 `strftime()` 函数，除了使用的语言环境数据来自 `locale` 表示的语言环境。

本地时区信息应设置为如同 `strftime()` 调用了 `tzset()`。

转换说明符

应支持以下转换说明符：

`%a` - 替换为语言环境的缩写工作日名称。[`tm_wday`]

`%A` - 替换为语言环境的完整工作日名称。[`tm_wday`]

`%b` - 替换为语言环境的缩写月份名称。[`tm_mon`]

`%B` - 替换为语言环境的完整月份名称。[`tm_mon`]

`%c` - 替换为语言环境的适当日期和时间表示。

`%C` - 替换为年份除以 100 并截断为整数的十进制数。[`tm_year`]

- 如果未指定最小字段宽度：

- 如果年份在 0 到 9999 之间（含边界值），应将两个字符放入 `s` 指向的数组中，如果本应只有一个数字则包含前导 '0'。

- [CX] 如果年份小于 0 或大于 9999，放入 `s` 指向的数组的字符数应为年份除以 100 并截断后的结果中的数字位数和前导符号字符（如果有），或两个，以较大者为准。

`%d` - 替换为月份中的日期，十进制数 [01,31]。[`tm_mday`]

`%D` - 等同于 `%m/%d/%y`。[`tm_mon` , `tm_mday` , `tm_year`]

%e - 替换为月份中的日期，十进制数 [1,31]；单个数字前面加空格。

[`tm_mday`]

%F - 如果没有标志且没有指定最小字段宽度，则等同于 %Y-%m-%d。（对于 1000 到 9999 年之间的年份，这提供 ISO 8601:2019 标准的完整表示，扩展格式的特定日期表示。）[`tm_year` , `tm_mon` , `tm_mday`]

%g - 替换为基于周的年份的最后两位数字，十进制数 [00,99]。[`tm_year` , `tm_wday` , `tm_yday`]

%G - 替换为基于周的年份，十进制数（例如 1977）。[`tm_year` , `tm_wday` , `tm_yday`]

%h - 等同于 %b。[`tm_mon`]

%H - 替换为小时（24小时制），十进制数 [00,23]。[`tm_hour`]

%I - 替换为小时（12小时制），十进制数 [01,12]。[`tm_hour`]

%j - 替换为年份中的第几天，十进制数 [001,366]。[`tm_yday`]

%m - 替换为月份，十进制数 [01,12]。[`tm_mon`]

%M - 替换为分钟，十进制数 [00,59]。[`tm_min`]

%n - 替换为换行符。

%p - 替换为语言环境中 a.m. 或 p.m. 的等价形式。[`tm_hour`]

%r - 替换为12小时制表示法的时间；[CX] 如果语言环境中不支持12小时制格式，这应该是空字符串或24小时制表示法的时间。在POSIX语言环境中，这应等同于 %I:%M:%S %p。[`tm_hour` , `tm_min` , `tm_sec`]

%R - 替换为24小时制表示法的时间（%H:%M）。[`tm_hour` , `tm_min`]

%s - [CX] 替换为自纪元以来的秒数，十进制数，按 `mktime()` 描述的计算方式计算。[`tm_year` , `tm_mon` , `tm_mday` , `tm_hour` , `tm_min` , `tm_sec` , `tm_isdst`]

%S - 替换为秒，十进制数 [00,60]。[`tm_sec`]

%t - 替换为制表符。

%T - 替换为时间（%H:%M:%S）。[`tm_hour` , `tm_min` , `tm_sec`]

%u - 替换为工作日，十进制数 [1,7]，1 表示星期一。[`tm_wday`]

%U - 替换为年份中的周数，十进制数 [00,53]。一月的第一个星期日是第1周的第一天；新年中此日期之前的日期在第0周。[`tm_year` , `tm_wday` , `tm_yday`]

%V - 替换为年份中的周数（星期一为一周的第一天），十进制数 [01,53]。如果包含1月1日的那一周在新年中有四天或更多天，则它被认为是第1周。否则，它是上一年的最后一周，下一周是第1周。1月4日和1月的第一个星期四总是在第1周。[`tm_year` , `tm_wday` , `tm_yday`]

%W - 替换为工作日，十进制数 [0,6]，0 表示星期日。[`tm_wday`]

%W - 替换为年份中的周数，十进制数 [00,53]。一月的第一个星期一是第1周的第一天；新年中此日期之前的日期在第0周。[`tm_year` , `tm_wday` , `tm_yday`]

%X - 替换为语言环境的适当日期表示。

%X - 替换为语言环境的适当时间表示。

%y - 替换为年份的最后两位数字，十进制数 [00,99]。[`tm_year`]

%Y - 替换为年份，十进制数（例如1997）。[`tm_year`]

%z - 替换为与UTC的偏移量，ISO 8601:2019 标准格式（+hhmm 或 -hhmm），如果无法确定时区则不替换任何字符。例如，"-0430" 表示比UTC晚4小时30分钟（格林威治以西）。[CX] 如果 `tm_isdst` 为零，则使用标准时间偏移量。如果 `tm_isdst` 大于零，则使用夏令时偏移量。如果 `tm_isdst` 为负数，则不返回任何字符。[`tm_isdst` , [CX] `tm_gmtoff`]

%Z - 替换为时区名称或缩写，如果不存在时区信息则不替换任何字节。
[`tm_isdst` , [CX] `tm_zone`]

%% - 替换为 %。

如果转换说明符与上述任何一个都不对应，则行为未定义。

[CX] 如果 `struct tm` 分解时间结构由 `localtime()` 或 `localtime_r()` 创建，或由 `mktime()` 修改，并且 `TZ` 的值随后被修改，则当 `strftime()` 使用这样的分解时间结构调用时，%Z 和 %z `strftime()` 转换说明符的结果未定义。

如果 `struct tm` 分解时间结构由 `gmtime()` 或 `gmtime_r()` 创建或修改，则当 `strftime()` 使用这样的分解时间结构调用时，%Z 和 %z 转换说明符的结果应指UTC还是当前本地时区未指定。

修改的转换说明符

某些转换说明符可以通过 E 或 O 修饰符字符修改，以指示应使用替代格式或规范，而不是未修改转换说明符通常使用的格式或规范。如果当前语言环境不存在替代格式或规范，则行为应等同于使用未修改的转换说明符。

%Ec - 替换为语言环境的替代适当日期和时间表示。

%EC - 替换为语言环境替表示中的基准年份（周期）名称。

%Ex - 替换为语言环境的替代日期表示。

%EX - 替换为语言环境的替代时间表示。

%Ey - 替换为语言环境替表示中与 %EC 的偏移量（仅年份）。

%EY - 替换为完整的替代年份表示。

%Ob - [CX] 替换为语言环境的缩写替代月份名称。

%OB - [CX] 替换为语言环境的适当替代完整月份名称。

%Od - 替换为月份中的日期，使用语言环境的替代数字符号，如果存在零的替代符号则根据需要用前导零填充；否则用前导空格字符填充。

%Oe - 替换为月份中的日期，使用语言环境的替代数字符号，根据需要用前导空格字符填充。

%OH - 使用语言环境的替代数字符号替换小时（24小时制）。

%OI - 使用语言环境的替代数字符号替换小时（12小时制）。

%Om - 使用语言环境的替代数字符号替换月份。

%OM - 使用语言环境的替代数字符号替换分钟。

%OS - 使用语言环境的替代数字符号替换秒。

%Ou - 使用语言环境的替表示替换工作日为数字（星期一=1）。

%OU - 使用语言环境的替代数字符号替换年份中的周数（星期日为一周的第一天，规则对应%U）。

%OV - 使用语言环境的替代数字符号替换年份中的周数（星期一为一周的第一天，规则对应%V）。

%Ow - 使用语言环境的替代数字符号替换工作日编号（星期日=0）。

%OW - 使用语言环境的替代数字符号替换年份中的周数（星期一为一周的第一天）。

%Oy - 使用语言环境的替代数字符号替换年份（与 %C 的偏移量）。

%g、%G 和 %V 根据 ISO 8601:2019 标准的基于周的年份给出值。在此系统中，周从星期一开始，年份的第1周是包含1月4日的那一周，该周也是包含年份第一个星期四的那一周，也是年份中至少包含四天的第一周。如果1月的第一个星期一是2日、3日或4日，则前面的几天是前一年的最后一周的一部分；因此，对于1999年1月2日星期六，%G 被替换为1998，%V 被替换为53。如果12月29日、30日或31日是星期一，它和任何后续日期是下一年的第1周的一部分。因此，对于1997年12月30日星期二，%G 被替换为1998，%V 被替换为01。

[CX] 如果 `strftime_l()` 的 `locale` 参数是特殊语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄，则行为未定义。

返回值

如果成功，这些函数应返回放入 `s` 指向数组的字节数，不包括终止的 NUL 字符。[CX] 如果成功，`errno` 不应更改。否则，应返回0，[CX] `errno` 应设置为指示错误，数组的内容未指定。

错误

[CX] 这些函数在以下情况下可能会失败：

- **[ERANGE]** - 结果的总字节数（包括终止的 NUL 字符）超过 `maxsize`。

这些函数在以下情况下可能会失败：

- **[EINVAL]** - `format` 字符串包含 %s 转换且自纪元以来的秒数为负数。
- **[EOVERFLOW]** - `format` 字符串包含 %s 转换且自纪元以来的秒数无法在 `time_t` 中表示。

示例

获取本地化日期字符串

以下示例首先将语言环境设置为用户的默认值。语言环境信息将在 `nl_langinfo()` 和 `strftime()` 函数中使用。`nl_langinfo()` 函数返回本地化的日期字符串，该字符串指定日期的布局方式。`strftime()` 函数使用此信息，并使用 `tm` 结构的值，将日期和时间信息放入 `datestring`。

```
#include <time.h>
#include <locale.h>
#include <langinfo.h>
...
struct tm *tm;
char datestring[256];
...
setlocale (LC_ALL, "");
...
strftime (datestring, sizeof(datestring), nl_langinfo (D_T_FMT)
```

应用程序用法

如果格式是空字符串或由转换规范（如 %p 或 %Z）组成，这些转换规范在当前语言环境中或由于 `tzname[]` 的当前设置而替换为无字符，则返回值 0 可能表示成功或失败。为了区分 `strftime()` 返回 0 时的成功和失败，应用程序可以在调用 `strftime()` 之前将 `errno` 设置为 0，并测试之后的 `errno` 是否为 0。

%S 的值范围是 [00,60] 而不是 [00,59]，以允许偶尔的闰秒。

某些转换说明符是其他说明符的重复。它们包含是为了与第 2 期发布的 `nl_cftime()` 和 `nl_ascftime()` 兼容。

`strftime()` 中的 %C、%F、%G 和 %Y 格式说明符总是打印完整值，但 `strptime()` %C、%F 和 %Y 格式说明符只扫描两位数字（假定为四位年份的前两位数字）对应 %C，和四位数字（假定为整个（四位）年份）对应 %F 和 %Y。这模仿了 `printf()` 和 `scanf()` 的行为。

在 C 或 POSIX 语言环境中，E 和 O 修饰符被忽略，以下说明符的替换字符串为：

- %a - %A 的前三个字符。
- %A - Sunday、Monday、...、Saturday 之一。
- %b - %B 的前三个字符。
- %B - January、February、...、December 之一。
- %c - 等同于 %a %b %e %T %Y。
- %p - AM 或 PM 之一。
- %r - 等同于 %I:%M:%S %p。
- %x - 等同于 %m/%d/%y。
- %X - 等同于 %T。
- %Z - 实现定义。

另请参阅

`asctime()`、`clock()`、`ctime()`、`difftime()`、`futimens()`、
`getdate()`、`gmtime()`、`localtime()`、`mktimes()`、`strptime()`、
`time()`、`tzset()`、`useLocale()`

XBD 7.3.5 LC_TIME、`<time.h>`

1.235. `strlen`, `strnlen` — 获取固定大小字符串的长度

概要

```
#include <string.h>

size_t strlen(const char *s);

[CX] size_t strnlen(const char *s, size_t maxlen);
```

描述

对于 `strlen()`：[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`strlen()` 函数应计算 `s` 指向的字符串中的字节数，不包括终止的 NUL 字符。

[CX] `strnlen()` 函数应计算 `s` 指向的数组中的字节数（不包括任何终止 NUL 字符）与 `maxlen` 参数值中的较小者。`strnlen()` 函数决不应检查超过 `s` 指向的数组的前 `maxlen` 个字节。

[CX] `strlen()` 和 `strnlen()` 函数在有效输入时不应更改 `errno` 的设置。

返回值

`strlen()` 函数应返回 `s` 的长度；不保留任何返回值来指示错误。

[CX] `strnlen()` 函数应返回 `s` 指向的数组中第一个空字节之前的字节数，如果 `s` 在前 `maxlen` 个字节内包含空字节；否则，它应返回 `maxlen`。

错误

未定义任何错误。

示例

获取字符串长度

以下示例使用 `strlen()` 获取 `key` 和 `data` 字符串的长度，以设置它们的最大长度。

```
#include <string.h>
...
struct element {
    char *key;
    char *data;
};

char *key, *data;
int len;

*keylength = *datalength = 0;
...
if ((len = strlen(key)) > *keylength)
    *keylength = len;
if ((len = strlen(data)) > *datalength)
    *datalength = len;
...
```

应用用法

无。

基本原理

无。

未来方向

无。

参见

- `strlcat()`

- `wcslen()`

XBD `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

RETURN VALUE 部分更新为指明 `strlen()` 返回 `s` 的长度，而不是先前所述的 `s` 本身。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 添加了 `strnlen()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0344 [560]。

Issue 8

应用了 Austin Group Defect 448，添加了 `strlen()` 和 `strnlen()` 在有效输入时不更改 `errno` 设置的要求。

应用了 Austin Group Defect 986，将 `strlcat()` 添加到 SEE ALSO 部分。

1.236. `strncat` — 将一个字符串的部分内容连接到另一个字符串末尾

概要

```
#include <string.h>

char *strncat(char *restrict s1, const char *restrict s2, size_
```

描述

`strncat()` 函数应将 `s2` 指向的数组中的不超过 `n` 个字节（不附加 NUL 字符及其后续字节）附加到 `s1` 指向的字符串末尾。`s2` 的初始字节覆盖 `s1` 末尾的 NUL 字符。结果总是附加一个终止的 NUL 字符。如果在重叠的对象之间进行复制，则行为是未定义的。

`strncat()` 函数在有效输入时不应改变 `errno` 的设置。

返回值

`strncat()` 函数应返回 `s1`；没有保留返回值来指示错误。

错误

未定义错误。

示例

无。

应用程序用法

无。

原理

无。

未来方向

无。

参见

- `strcat()`
- `strlcat()`
- XBD `<string.h>`

变更历史

首次发布于 Issue 1。

源自 SVID 的 Issue 1。

Issue 6

`strncat()` 原型已更新，以与 ISO/IEC 9899:1999 标准对齐。

Issue 8

应用了 Austin Group 缺陷 448，增加了 `strncat()` 在有效输入时不改变 `errno` 设置的要求。

应用了 Austin Group 缺陷 986，在 SEE ALSO 部分添加了 `strlcat()`。

1.237. strncmp

SYNOPSIS (概要)

```
#include <string.h>

int strncmp(const char *s1, const char *s2, size_t n);
```

DESCRIPTION (描述)

[Option Start] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 本卷遵循 ISO C 标准。[Option End]

`strncmp()` 函数应当比较由 `s1` 指向的数组与由 `s2` 指向的数组中不超过 `n` 个字节的内容 (NUL 字符后的字节不参与比较)。

非零返回值的符号由被比较字符串中第一对不同字节 (两者都解释为 `unsigned char` 类型) 的差值符号决定。

[Option Start] 在有效输入时, `strncmp()` 函数不应改变 `errno` 的设置。
[Option End]

RETURN VALUE (返回值)

成功完成后, 如果 `s1` 指向的可能以 null 结尾的数组分别大于、等于或小于 `s2` 指向的可能以 null 结尾的数组, `strncmp()` 应返回一个大于、等于或小于 0 的整数。

ERRORS (错误)

未定义任何错误。

以下部分为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `strcmp()`
- XBD `<string.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

超出 ISO C 标准的扩展被标记出来。

Issue 8

应用了 Austin Group Defect 448，增加了在有效输入时 `strncpy()` 不改变 `errno` 设置的要求。

补充信息结束。

1.238. `strncpy`

SYNOPSIS

```
#include <string.h>

[CX] char *stpncpy(char *restrict s1, const char *restrict s2,
char *strncpy(char *restrict s1, const char *restrict s2, size_
```



DESCRIPTION

对于 `strncpy()`: [CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

[CX] `stpncpy()` 和 `strncpy()` 函数应从 `s2` 指向的数组向 `s1` 指向的数组复制不超过 `n` 个字节 (NUL 字符后的字节不复制)。

如果 `s2` 指向的数组是一个长度小于 `n` 字节的字符串，则应在 `s1` 指向的数组中的副本后附加 NUL 字符，直到总共写入 `n` 个字节。

如果复制操作发生在重叠的对象之间，则行为未定义。

[CX] `strncpy()` 和 `stpncpy()` 函数在有效输入时不应更改 `errno` 的设置。

RETURN VALUE

[CX] 如果向目标写入了 NUL 字符，`stpncpy()` 函数应返回第一个此类 NUL 字符的地址。否则，应返回 `&s1[n]`。

`strncpy()` 函数应返回 `s1`。

没有保留用于指示错误的返回值。

ERRORS

未定义错误。

以下章节为补充信息。

EXAMPLES

无。

APPLICATION USAGE

应用程序必须在 `s1` 中提供用于传输 `n` 个字节的空间，并确保 `s2` 和 `s1` 数组不重叠。

字符移动在不同实现中的执行方式不同。因此，重叠移动可能会产生意外结果。

如果 `s2` 指向的数组的前 `n` 个字节中没有 NUL 字符字节，则结果不是以 `null` 结尾的。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `strcpy()`
- `strlcat()`
- `wcsncpy()`

XBD

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

strncpy() 原型已更新，以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 1 添加了 stpncpy() 函数。

Issue 8

应用了 Austin Group Defect 448，添加了 strncpy() 和 stpncpy() 在有效输入时不更改 errno 设置的要求。

应用了 Austin Group Defect 986，在 SEE ALSO 章节中添加了 strlcat()。

补充信息结束。

1.239. `strupbrk`

SYNOPSIS

```
#include <string.h>

char *strupbrk(const char *s1, const char *s2);
```

DESCRIPTION

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`strupbrk()` 函数应定位 `s1` 所指向的字符串中第一次出现的 `s2` 所指向字符串中的任何字节。

[CX] `strupbrk()` 函数在有效输入上不应改变 `errno` 的设置。

RETURN VALUE

成功完成后, `strupbrk()` 应返回指向该字节的指针; 如果 `s2` 中的任何字节在 `s1` 中都没有出现, 则返回空指针。

ERRORS

未定义错误。

以下部分为参考信息。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `strchr()`
- `strrchr()`
- XBD `<string.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 448，增加了 `strpbrk()` 在有效输入上不改变 `errno` 设置的要求。

1.240. strrchr — 字符串扫描操作

概要

```
#include <string.h>

char *strrchr(const char *s, int c);
```

描述

`strrchr()` 函数应定位 `s` 所指向的字符串中 `c` (转换为 `char` 类型) 的最后一次出现。终止的 NUL 字符被视为字符串的一部分。

对于有效输入, `strrchr()` 函数不应更改 `errno` 的设置。

返回值

成功完成后, `strrchr()` 应返回一个指向该字节的指针, 如果 `c` 不在字符串中出现, 则返回空指针。

错误

未定义错误。

示例

获取文件的基础名称

以下示例使用 `strrchr()` 获取指向文件基础名称的指针。`strrchr()` 函数通过文件名称向后搜索, 以找到 `name` 中的最后一个 '/' 字符。这个指针 (加一) 将指向文件的基础名称。

```
#include <string.h>
...
const char *name;
char *basename;
...
```

```
basename = strrchr(name, '/') + 1;
```

```
...
```

应用用法

无。

基本原理

无。

未来方向

无。

参见

- [strchr\(\)](#)
- [<string.h>](#)

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 448，增加了一项要求：对于有效输入，[strrchr\(\)](#) 不得更改 [errno](#) 的设置。

1.241. `strspn` — 获取子字符串长度

概要

```
#include <string.h>

size_t strspn(const char *s1, const char *s2);
```

描述

`strspn()` 函数应计算由 `s1` 所指向的字符串的最大初始段的长度（以字节为单位），该段完全由来自 `s2` 所指向字符串的字节组成。

`strspn()` 函数在有效输入时不改变 `errno` 的设置。

返回值

`strspn()` 函数应返回计算的长度；没有保留返回值来指示错误。

错误

未定义错误。

应用用法

无。

原理

无。

未来方向

无。

参见

- `strcspn()`
- `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5: 更新了返回值部分，指明 `strspn()` 返回 `s` 的长度，而不是像之前所说的返回 `s` 本身。

Issue 7: 应用了 SD5-XSH-ERN-182。

Issue 8: 应用了 Austin Group Defect 448，添加了 `strspn()` 在有效输入时不改变 `errno` 设置的要求。

1.242. strstr — 查找子字符串

概要

```
#include <string.h>

char *strstr(const char *s1, const char *s2);
```

描述

[CX] 本参考页面所描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 版本遵循 ISO C 标准。

`strstr()` 函数应在 `s1` 所指向的字符串中定位 `s2` 所指向的字符串的字节序列（不包括终止的 NUL 字符）的第一次出现。

[CX] `strstr()` 函数在有效输入时不应改变 `errno` 的设置。

返回值

成功完成时，`strstr()` 应返回指向所定位字符串的指针；如果未找到该字符串，则返回空指针。

如果 `s2` 指向长度为零的字符串，则函数应返回 `s1`。

错误

未定义错误。

示例

无。

应用用法

无。

原理

无。

未来方向

无。

参见

- [memmem\(\)](#)
- [strchr\(\)](#)
- XBD [`<string.h>`](#)

变更历史

首次发布于 Issue 3。为与 ANSI C 标准保持一致而包含在内。

Issue 8

应用了 Austin Group Defect 448，增加了 [strstr\(\)](#) 在有效输入时不改变 [errno](#) 设置的要求。

应用了 Austin Group Defect 1061，在参见部分增加了 [memmem\(\)](#)。

1.243. strtod, strtod, strtold — 将字符串转换为双精度浮点数

概要

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr)
float strtod(const char *restrict nptr, char **restrict endptr)
long double strtold(const char *restrict nptr, char **restrict
```

描述

[CX] 本参考页描述的功能与ISO C标准保持一致。此处描述的要求与ISO C标准之间的冲突是无意的。本卷POSIX.1-2024遵循ISO C标准。

这些函数应将 `nptr` 指向的字符串的初始部分分别转换为 `double`、`float` 和 `long double` 表示。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 一个被解释为浮点常量或表示无穷大或NaN的主题序列
3. 由一个或多个未识别字符组成的最终字符串，包括输入字符串的终止NUL字符

然后它们应尝试将主题序列转换为浮点数，并返回结果。

主题序列的预期形式是一个可选的'+'或'-'符号，然后是以下形式之一：

- 一个非空的十进制数字序列，可选择包含基数字符；然后是一个可选的指数部分，由字符'e'或字符'E'组成，可选择后跟'+'或'-'字符，然后再后跟一个或多个十进制数字
- 一个0x或0X，然后是一个非空的十六进制数字序列，可选择包含基数字符；然后是一个可选的二进制指数部分，由字符'p'或字符'P'组成，可选择后跟'+'或'-'字符，然后再后跟一个或多个十进制数字
- INF或INFINITY之一，忽略大小写
- NAN或NAN(n-char-sequenceopt)之一，NAN部分忽略大小写，其中：

```
n-char-sequence:  
    digit  
    nondigit  
    n-char-sequence digit  
    n-char-sequence nondigit
```

主题序列定义为输入字符串的最长初始子序列，从第一个非空白字节开始，具有预期形式。如果输入字符串不具有预期形式，则主题序列不包含任何字符。

如果主题序列具有浮点数的预期形式，从第一个数字或小数点字符（以先出现的为准）开始的字符序列应被解释为C语言的浮点常量，除了基数字符应代替句点使用，并且如果在十进制浮点数中既没有指数部分也没有基数字符，或者在十六进制浮点数中没有二进制指数部分，则假定值零的适当类型指数部分跟在字符串中的最后一个数字之后。如果主题序列以开始，该序列应被解释为否定。字符序列INF或INFINITY应被解释为无穷大，如果在返回类型中可表示，否则应被解释为对于返回类型范围来说过大的浮点常量。字符序列NAN或NAN(n-char-sequenceopt)应被解释为安静NaN，如果在返回类型中支持，否则应被解释为不具有预期形式的主题序列部分；n-char序列的含义是实现定义的。指向最终字符串的指针存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果主题序列具有十六进制形式且FLT_RADIX是2的幂，则转换产生的值被正确舍入。

[CX] 基数字符在当前语言环境（类别LC_NUMERIC）中定义。在POSIX语言环境或基数字符未定义的语言环境中，基数字符应默认为 ('.')。

在C [CX]或POSIX语言环境之外，可以接受其他特定于语言环境的主题序列形式。

如果主题序列为空或不具有预期形式，则不执行转换；`nptr` 的值存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果成功，这些函数不应更改 `errno` 的设置。

由于错误时返回0，成功时也是有效的返回值，希望检查错误情况的应用程序应将 `errno` 设置为0，然后调用 `strtod()`、`strtof()` 或 `strtold()`，然后检查 `errno`。

返回值

成功完成时，这些函数应返回转换后的值。如果无法执行转换，应返回0，并且 `errno` 可以设置为[EINVAL]。

如果正确值会导致溢出且默认舍入生效，应返回 $\pm\text{HUGE_VAL}$ 、 $\pm\text{HUGE_VALF}$ 或 $\pm\text{HUGE_VALL}$ （根据值的符号），并且`errno`应设置为[ERANGE]。

如果正确值会导致下溢，应返回其大小不大于返回类型中最小规范化正数的值[CX]，并且`errno`设置为[ERANGE]。

错误

这些函数在以下情况下可能失败：

- [ERANGE] 要返回的值会导致溢出且默认舍入生效[CX]，或者要返回的值会导致下溢。

这些函数在以下情况下可能失败：

- [EINVAL] [CX] 无法执行转换。

示例

无。

应用程序用法

如果主题序列具有十六进制形式且`FLT_RADIX`不是2的幂，且结果不能精确表示，结果应该是适当内部格式中与十六进制浮点源值相邻的两个数字之一，额外要求误差对于当前舍入方向应该具有正确的符号。

如果主题序列具有十进制形式且最多有`DECIMAL_DIG`（在中定义）个有效数字，结果应该被正确舍入。如果主题序列D具有十进制形式且有超过`DECIMAL_DIG`个有效数字，考虑两个边界相邻的十进制字符串L和U，两者都具有`DECIMAL_DIG`个有效数字，使得L、D和U的值满足 $L \leq D \leq U$ 。结果应该是通过根据当前舍入方向正确舍入L和U获得的（相等或相邻）值之一，额外要求相对于D的误差对于当前舍入方向应该具有正确的符号。

ISO/IEC 9899:1999 标准引入的 `strtod()` 更改可能改变符合 ISO/IEC 9899:1990 标准及本标准早期版本的良好格式应用程序的行为。一个这样的例子是：

```
int
what_kind_of_number (char *s)
{
    char *endp;
    double d;
```

```
long l;

d = strtod(s, &endp);
if (s != endp && *endp == '\0')
    printf("It's a float with value %g\n", d);
else
{
    l = strtol(s, &endp, 0);
    if (s != endp && *endp == '\0')
        printf("It's an integer with value %ld\n", l);
    else
        return 1;
}
return 0;
}
```

如果使用以下方式调用函数：

```
what_kind_of_number ("0x10")
```

符合ISO/IEC 9899:1990标准的库将导致函数打印：

```
It's an integer with value 16
```

使用ISO/IEC 9899:1999标准，结果是：

```
It's a float with value 16
```

行为的变化是由于包含了十六进制表示法的浮点数，而不要求小数点或二进制指数存在。

原理

无。

未来方向

无。

另请参阅

- [fscanf\(\)](#)

- `isspace()`
- `localeconv()`
- `setlocale()`
- `strtol()`
- XBD 7. 语言环境
- `<float.h>`
- `<stdlib.h>`

变更历史

首次发布于Issue 1

派生自SVID的Issue 1。

Issue 5

描述更新为指示如果函数成功则不更改 `errno`。

Issue 6

超出ISO C标准的扩展被标记。

以下对POSIX实现的新要求源于与单一UNIX规范的对齐：

- 在返回值和错误部分，如果无法执行转换，则添加[EINVAL]可选错误条件。

为与ISO/IEC 9899:1999标准对齐进行以下更改：

- `strtod()` 函数更新。
- 添加 `strtod()` 和 `strtold()` 函数。
- 描述广泛修订。

纳入ISO/IEC 9899:1999标准，技术勘误1。

应用IEEE Std 1003.1-2001/Cor 1-2002，项目XSH/TC1/D6/61，修正返回值部分的第二段。此更改阐明了返回值的符号。

Issue 7

应用Austin Group解释1003.1-2001 #015。

应用 POSIX.1-2008 , 技术勘误 1 , XSH/TC1-2008/0610 [302] , XSH/TC1-2008/0611 [94] , 和 XSH/TC1-2008/0612 [105]。

应用 POSIX.1-2008 , 技术勘误 2 , XSH/TC2-2008/0348 [584] 和 XSH/TC2-2008/0349 [796]。

Issue 8

应用Austin Group缺陷1163, 阐明输入字符串中空白字符的处理。

应用Austin Group缺陷1213, 修正应用程序用法部分中的一些印刷错误。

应用Austin Group缺陷1302, 使这些函数与ISO/IEC 9899:2018标准对齐。

应用Austin Group缺陷1686, 为返回值部分中的一些文本添加CX阴影。

1.244. strtod, strtod, strtold - 将字符串转换为双精度浮点数

概要

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr)
float strtodf(const char *restrict nptr, char **restrict endptr)
long double strtold(const char *restrict nptr, char **restrict
```

描述

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 标准遵循 ISO C 标准。

这些函数应将 **nptr** 指向的字符串的初始部分分别转换为 **double**、**float** 和 **long double** 表示形式。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 一个解释为浮点常量或表示无穷大或 NaN 的主题序列
3. 由一个或多个未识别字符组成的最终字符串，包括输入字符串的终止 NUL 字符

然后它们应尝试将主题序列转换为浮点数，并返回结果。

主题序列的预期形式是一个可选的 '+' 或 '-' 符号，然后是以下形式之一：

- 一个非空的十进制数字序列，可选择包含基数字符；然后是一个可选的指数部分，由字符 'e' 或字符 'E' 组成，可选择后跟 '+' 或 '-' 字符，再后跟一个或多个十进制数字
- 一个 0x 或 0X，然后是一个非空的十六进制数字序列，可选择包含基数字符；然后是一个可选的二进制指数部分，由字符 'p' 或字符 'P' 组成，可选择后跟 '+' 或 '-' 字符，再后跟一个或多个十进制数字
- INF 或 INFINITY 之一，忽略大小写
- NAN 或 NAN(*n-char-sequenceopt*) 之一，在 NAN 部分忽略大小写，其中：

```
n-char-sequence:  
    digit  
    nondigit  
    n-char-sequence digit  
    n-char-sequence nondigit
```

主题序列定义为输入字符串的最长初始子序列，从第一个非空白字节开始，具有预期形式。如果输入字符串不符合预期形式，则主题序列不包含任何字符。

如果主题序列具有浮点数的预期形式，则从第一个数字或小数点字符（以先出现的为准）开始的字符序列应解释为 C 语言的浮点常量，但使用基数字符代替句点，如果在十进制浮点数中既没有指数部分也没有基数字符，或者在十六进制浮点数中没有二进制指数部分，则假定在字符串中的最后一个数字后跟一个值为零的适当类型的指数部分。如果主题序列以 `l` 开始，则该序列应解释为被否定。字符序列 `INF` 或 `INFINITY` 应解释为无穷大，如果在返回类型中可表示，否则解释为超出返回类型范围的过大浮点常量。字符序列 `NAN` 或 `NAN(n-char-sequence)` 应解释为安静 `NaN`，如果在返回类型中支持，否则解释为不符合预期形式的主题序列部分；*n-char* 序列的含义由实现定义。指向最终字符串的指针存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果主题序列具有十六进制形式且 `FLT_RADIX` 是 2 的幂，则转换产生的值是正确舍入的。

[CX] 基数字符在当前区域设置（`LC_NUMERIC` 类别）中定义。在 POSIX 区域设置中，或在基数字符未定义的区域设置中，基数字符应默认为 `('.'`）。

在 C [CX] 或 POSIX 区域设置之外，可以接受其他特定于区域设置的主题序列形式。

如果主题序列为空或不符合预期形式，则不执行转换；`nptr` 的值存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果成功，这些函数不应更改 `errno` 的设置。

由于错误时返回 0，成功时也可能返回 0，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strtod()`、`strtof()` 或 `strtold()`，然后检查 `errno`。

返回值

成功完成后，这些函数应返回转换后的值。如果无法执行转换，则应返回 0，并可能将 `errno` 设置为 `[EINVAL]`。

如果正确值会导致溢出且默认舍入生效，则应返回 `±HUGE_VAL`、`±HUGE_VALF` 或 `±HUGE_VALL`（根据值的符号），并应将 `errno` 设置为

[ERANGE]。

如果正确值会导致下溢，则应返回其绝对值不大于返回类型中最小归一化正数的值 [CX]，并将 `errno` 设置为 [ERANGE]。

错误

这些函数在以下情况下应失败：

[ERANGE]

要返回的值会导致溢出且默认舍入生效 [CX]，或要返回的值会导致下溢。

这些函数在以下情况下可能失败：

[EINVAL]

[CX] 无法执行转换。

以下部分为信息性内容。

示例

无。

应用程序用法

如果主题序列具有十六进制形式且 `FLT_RADIX` 不是 2 的幂，且结果不能精确表示，则结果应是与十六进制浮点源值相邻的两个适当内部格式数字之一，附加条件是对于当前舍入方向，误差应具有正确的符号。

如果主题序列具有十进制形式且最多有 `DECIMAL_DIG`（在 `<float.h>` 中定义）个有效数字，则结果应正确舍入。如果主题序列 `D` 具有十进制形式且有超过 `DECIMAL_DIG` 个有效数字，考虑两个边界、相邻的十进制字符串 `L` 和 `U`，两者都有 `DECIMAL_DIG` 个有效数字，使得 `L`、`D` 和 `U` 的值满足 `L <= D <= U`。结果应是通过根据当前舍入方向正确舍入 `L` 和 `U` 获得的（相等或相邻）值之一，附加条件是相对于 `D` 的误差对于当前舍入方向应具有正确的符号。

ISO/IEC 9899:1999 标准引入的对 `strtod()` 的更改可能会改变符合 ISO/IEC 9899:1990 标准及本标准早期版本的应用程序的行为。一个这样的示例是：

```
int
what_kind_of_number (char *s)
```

```
{  
    char *endp;  
    double d;  
    long l;  
  
    d = strtod(s, &endp);  
    if (s != endp && *endp == '\0')  
        printf("It's a float with value %g\n", d);  
    else  
    {  
        l = strtol(s, &endp, 0);  
        if (s != endp && *endp == '\0')  
            printf("It's an integer with value %ld\n", l);  
        else  
            return 1;  
    }  
    return 0;  
}
```

如果使用以下调用：

```
what_kind_of_number ("0x10")
```

符合 ISO/IEC 9899:1990 标准的库将导致函数打印：

```
It's an integer with value 16
```

使用 ISO/IEC 9899:1999 标准，结果是：

```
It's a float with value 16
```

行为的改变是由于包含了十六进制表示法的浮点数，而不要求存在小数点或二进制指数。

原理

无。

未来方向

无。

参见

`fscanf()`, `isspace()`, `localeconv()`, `setlocale()`, `strtol()`

XBD 7. Locale, `<float.h>`, `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

描述更新为指出如果函数成功，则不更改 `errno`。

Issue 6

标记了超出 ISO C 标准的扩展。

对 POSIX 实现的以下新要求源自与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 和 ERRORS 部分，如果无法执行转换，则添加 [EINVAL] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐进行了以下更改：

- 更新了 `strtod()` 函数。
- 添加了 `strtod()` 和 `strtold()` 函数。
- 大幅修订了 DESCRIPTION。

纳入了 ISO/IEC 9899:1999 标准，技术勘误 1。

应用了 IEEE Std 1003.1-2001/Cor 1-2002，项目 XSH/TC1/D6/61，更正了 RETURN VALUE 部分的第二段。此更改阐明了返回值的符号。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #015。

应用了 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0610 [302]、XSH/TC1-2008/0611 [94] 和 XSH/TC1-2008/0612 [105]。

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0348 [584] 和 XSH/TC2-2008/0349 [796]。

Issue 8

应用了 Austin Group Defect 1163，阐明了输入字符串中空白的处理。

应用了 Austin Group Defect 1213，更正了 APPLICATION USAGE 部分中的一些印刷错误。

应用了 Austin Group Defect 1302，使这些函数与 ISO/IEC 9899:2018 标准对齐。

应用了 Austin Group Defect 1686，在 RETURN VALUE 部分的一些文本中添加了 CX 阴影。

1.245. `strtoimax`, `strtoumax` — 将字符串转换为整数类型

概要

```
#include <inttypes.h>

intmax_t strtoimax(const char *restrict nptr,
                    char **restrict endptr,
                    int base);

uintmax_t strtoumax(const char *restrict nptr,
                     char **restrict endptr,
                     int base);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

这些函数应等效于 `strtol()`、`strtoll()`、`strtoul()` 和 `strtoull()` 函数，不同之处在于字符串的初始部分应分别转换为 `intmax_t` 和 `uintmax_t` 表示。

返回值

这些函数应返回转换后的值（如果有）。

如果无法执行转换，应返回零 [CX] 且 `errno` 可设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0 且 `errno` 应设置为 [EINVAL]。

如果正确的值超出可表示值的范围，应返回 {INTMAX_MAX}、{INTMAX_MIN} 或 {UINTMAX_MAX}（根据返回类型和值的符号，如果有），且 `errno` 应设置为 [ERANGE]。

错误

这些函数在以下情况下应失败：

- [EINVAL] [CX] `base` 的值不受支持。
- [ERANGE] 要返回的值不可表示。

这些函数在以下情况下可能失败：

- [EINVAL] 无法执行转换。
-

以下部分为参考信息。

示例

无。

应用程序用法

由于如果 `base` 的值不受支持，`*endptr` 的值是未指定的，应用程序应在调用前确保 `base` 具有支持的值（0 或 2 到 36 之间），或者在检查 `*endptr` 之前检查 [EINVAL] 错误。

基本原理

无。

未来方向

无。

参见

- `strtol()`
- `strtoul()`
- XBD `<inttypes.h>`

变更历史

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0613 [453]
和 XSH/TC1-2008/0614 [453]。

1.246. strtok, strtok_r — 将字符串分割为标记

概要

```
#include <string.h>

char *strtok(char *restrict s, const char *restrict sep);

[X] char *strtok_r(char *restrict s, const char *restrict sep,
                    char **restrict state);
```

描述

对于 `strtok()`：

[X] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

一系列对 `strtok()` 的调用将 `s` 指向的字符串分解为一系列标记 (token)，每个标记由 `sep` 指向的字符串中的一个字节分隔。序列中的第一次调用以 `s` 作为第一个参数，后续调用以空指针作为第一个参数。`sep` 指向的分隔符字符串在每次调用中可以不同。

序列中的第一次调用在 `s` 指向的字符串中搜索第一个不包含在当前 `sep` 指向的分隔符字符串中的字节。如果找不到这样的字节，那么 `s` 指向的字符串中没有标记，`strtok()` 应返回空指针。如果找到这样的字节，它就是第一个标记的开始。

`strtok()` 函数然后从该位置开始搜索包含在当前分隔符字符串中的字节。如果找不到这样的字节，当前标记延伸到 `s` 指向的字符串的末尾，后续对标记的搜索应返回空指针。如果找到这样的字节，它会被 NUL 字符覆盖，从而终止当前标记。`strtok()` 函数保存指向下一个字节的指针，下一次搜索标记将从此开始。

每个后续调用（以空指针作为第一个参数的值）都从保存的指针开始搜索，并如上所述行为。

实现的行为应如同本 POSIX.1-2024 卷中定义的任何函数都不会调用 `strtok()`。

`strtok()` 函数不需要是线程安全的；但是，`strtok()` 应避免与所有其他函数发生数据竞争。

[X] `strtok_r()` 函数应等同于 `strtok()`，不同之处在于 `strtok_r()` 应是线程安全的，并且参数 `state` 指向一个用户提供的指针，允许 `strtok_r()` 在扫描同一字符串的调用之间维护状态。应用程序应确保 `state` 指向的指针对于由 `strtok_r()` 调用并发处理的每个字符串 (`s`) 是唯一的。应用程序不需要将 `state` 指向的指针初始化为任何特定值。实现不应将 `state` 指向的指针更新为指向（直接或间接）字符串 `s` 之外的、需要调用者释放或释放的资源。

[X] 在有效输入上，`strtok()` 和 `strtok_r()` 函数不应更改 `errno` 的设置。

返回值

成功完成后，`strtok()` 应返回指向标记第一个字节的指针。否则，如果没有标记，`strtok()` 应返回空指针。

[X] `strtok_r()` 函数应返回指向找到的标记的指针，当找不到标记时返回空指针。

错误

未定义错误。

示例

搜索单词分隔符

以下示例搜索由空格字符分隔的标记。

```
#include <string.h>
...
char *token;
char line[] = "LINE TO BE SEPARATED";
char *search = " ";

/* Token 将指向 "LINE"。 */
token = strtok(line, search);

/* Token 将指向 "TO"。 */
token = strtok(NULL, search);
```

在缓冲区中查找前两个字段

以下示例使用 `strtok()` 查找由空格、制表符或换行符的任意组合分隔的两个字符串（一个键和与该键关联的数据），这些分隔符位于 `buffer` 指向的字符数组的开始处。

```
#include <string.h>
...
char *buffer;
...
struct element {
    char *key;
    char *data;
} e;
...
// 加载缓冲区...
...
// 获取键及其数据...
e.key = strtok(buffer, " \t\n");
e.data = strtok(NULL, " \t\n");
// 处理缓冲区的其余内容...
...
```

应用程序用法

请注意，如果 `sep` 是空字符串，`strtok()` 和 `strtok_r()` 会返回指向正在被标记化的字符串剩余部分的指针。

`strtok_r()` 函数是线程安全的，并将其状态存储在用户提供的缓冲区中，而不是可能使用静态数据区域，该区域可能被来自另一个线程的不相关调用覆盖。

基本原理

`strtok()` 函数在较大的字符串中搜索分隔符字符串。它返回指向分隔符字符串之间最后一个子字符串的指针。此函数使用静态存储来跟踪调用之间的当前字符串位置。新函数 `strtok_r()` 接受一个额外的参数 `state`，用于跟踪字符串中的当前位置。

未来方向

无。

另请参阅

XBD `<string.h>`

变更历史

首次在 Issue 1 中发布。派生自 SVID 的 Issue 1。

Issue 5

包含 `strtok_r()` 函数以与 POSIX 线程扩展对齐。

在描述中添加了一个注释，表明 `strtok()` 函数不需要是可重入的。

Issue 6

标记了超出 ISO C 标准的扩展。

`strtok_r()` 函数被标记为线程安全函数选项的一部分。

在描述中，关于可重入性的注释被扩展以涵盖线程安全性。

应用程序用法部分被更新，包含关于线程安全函数及其避免可能使用静态数据区域的注释。

为了与 ISO/IEC 9899:1999 标准对齐，向 `strtok()` 和 `strtok_r()` 原型添加了 `restrict` 关键字。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #156。

应用了 SD5-XSH-ERN-235，更正了一个示例。

`strtok_r()` 函数从线程安全函数选项移动到基本规范。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0615 [177]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0350 [878]。

Issue 8

应用了 Austin Group Defect 448，添加了 `strtok()` 和 `strtok_r()` 在有效输入上不更改 `errno` 设置的要求。

应用了 Austin Group Defect 1302，使 `strtok()` 函数与 ISO/IEC 9899:2018 标准对齐。

1.247. strtok, strtok_r — 将字符串分割为标记

概要

```
#include <string.h>

char *strtok(char *restrict s, const char *restrict sep);

[CX] char *strtok_r(char *restrict s, const char *restrict sep,
                     char **restrict state);
```

描述

对于 `strtok()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵循 ISO C 标准。

一系列对 `strtok()` 的调用将 `s` 指向的字符串分解为一系列标记，每个标记由 `sep` 指向的字符串中的一个字节分隔。序列中的第一次调用以 `s` 作为第一个参数，随后的调用则以空指针作为第一个参数。`sep` 指向的分隔符字符串可以在不同调用间有所差异。

序列中的第一次调用在 `s` 指向的字符串中搜索第一个不包含在当前 `sep` 指向的分隔符字符串中的字节。如果找不到这样的字节，那么 `s` 指向的字符串中没有标记，`strtok()` 应返回空指针。如果找到这样的字节，它就是第一个标记的开始。

`strtok()` 函数然后从该位置开始搜索一个包含在当前分隔符字符串中的字节。如果找不到这样的字节，当前标记扩展到 `s` 指向的字符串的末尾，后续对标记的搜索应返回空指针。如果找到这样的字节，它会被一个 NUL 字符覆盖，从而终止当前标记。`strtok()` 函数保存一个指向后续字节的指针，下一次搜索标记将从该指针开始。

每次后续调用都以空指针作为第一个参数的值，从保存的指针开始搜索，并按上述方式运行。

实现的行为应该如同本卷 POSIX.1-2024 中定义的任何函数都不调用 `strtok()`。

`strtok()` 函数不需要是线程安全的；但是，`strtok()` 应避免与所有其他函数发生数据竞争。

[CX] `strtok_r()` 函数应等价于 `strtok()`，但 `strtok_r()` 应该是线程安全的，并且参数 `state` 指向一个用户提供的指针，允许 `strtok_r()` 在扫描同一字符串的调用之间维护状态。应用程序应确保 `state` 指向的指针对于每个正在被 `strtok_r()` 调用并发处理的字符串 (`s`) 是唯一的。应用程序不需要将 `state` 指向的指针初始化为任何特定值。实现不应将 `state` 指向的指针更新为指向（直接或间接）字符串 `s` 之外的、需要调用者释放或释放的资源。

[CX] `strtok()` 和 `strtok_r()` 函数不应在有效输入时改变 `errno` 的设置。

返回值

成功完成后，`strtok()` 应返回指向标记第一个字节的指针。否则，如果没有标记，`strtok()` 应返回空指针。

[CX] `strtok_r()` 函数应返回指向找到的标记的指针，当没有找到标记时返回空指针。

错误

未定义错误。

示例

搜索单词分隔符

以下示例搜索由空格字符分隔的标记。

```
#include <string.h>
...
char *token;
char line[] = "LINE TO BE SEPARATED";
char *search = " ";

/* Token 将指向 "LINE"。 */
token = strtok(line, search);

/* Token 将指向 "TO"。 */
token = strtok(NULL, search);
```

在缓冲区中查找前两个字段

以下示例使用 `strtok()` 查找由空格、制表符或换行符的任意组合分隔的两个字符串（一个键和与该键关联的数据），这些分隔符位于 `buffer` 指向的字符数组的开始位置。

```
#include <string.h>
...
char    *buffer;
...
struct element {
    char *key;
    char *data;
} e;
...
// 加载缓冲区...
...
// 获取键及其数据...
e.key = strtok(buffer, " \t\n");
e.data = strtok(NULL, " \t\n");
// 处理缓冲区的其余内容...
...
```

应用程序用法

请注意，如果 `sep` 是空字符串，`strtok()` 和 `strtok_r()` 返回指向被标记化字符串剩余部分的指针。

`strtok_r()` 函数是线程安全的，其状态存储在用户提供的缓冲区中，而不是可能被来自另一个线程的不相关调用覆盖的静态数据区域。

原理说明

`strtok()` 函数在较大字符串中搜索分隔符字符串。它返回指向分隔符字符串之间最后一个子字符串的指针。该函数使用静态存储来跟踪调用之间的当前字符串位置。新函数 `strtok_r()` 接受一个额外的参数 `state`，用于跟踪字符串中的当前位置。

未来方向

无。

参见

XBD `<string.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

包含 `strtok_r()` 函数以与 POSIX 线程扩展保持一致。

添加了一个说明，指出 `strtok()` 函数不需要是可重入的。

Issue 6

超出了 ISO C 标准的扩展被标记。

`strtok_r()` 函数被标记为线程安全函数选项的一部分。

在描述中，关于可重入性的说明被扩展以涵盖线程安全性。

应用程序用法部分被更新，包括关于线程安全函数及其避免可能使用静态数据区域的说明。

为了与 ISO/IEC 9899:1999 标准保持一致，向 `strtok()` 和 `strtok_r()` 原型添加了 `restrict` 关键字。

Issue 7

应用了 Austin Group 解释 1003.1-2001 #156。

应用了 SD5-XSH-ERN-235，修正了一个示例。

`strtok_r()` 函数从线程安全函数选项移动到基础标准。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0615 [177]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0350 [878]。

Issue 8

应用了 Austin Group 缺陷 448，添加了 `strtok()` 和 `strtok_r()` 在有效输入时不改变 `errno` 设置的要求。

应用了 Austin Group 缺陷 1302，使 `strtok()` 函数与 ISO/IEC 9899:2018 标准保持一致。

1.248. strtol, strtoll — 将字符串转换为长整型

概要

```
#include <stdlib.h>

long strtol(const char *restrict nptr, char **restrict endptr,
long long strtoll(const char *restrict nptr, char **restrict en
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的冲突是无意的。POSIX.1-2024 标准服从 ISO C 标准。

这些函数应将 `nptr` 指向的字符串的初始部分分别转换为 `long` 和 `long long` 类型的表示。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 被解释为整数的主题序列，该整数的基数由 `base` 的值确定
3. 一个包含一个或多个无法识别字符的最终字符串，包括输入字符串的终止 NUL 字符。

然后它们应尝试将主题序列转换为整数，并返回结果。

如果 `base` 的值为 0，主题序列的预期形式为十进制常量、八进制常量或十六进制常量，其中任何一个都可以前面加 '+' 或 '-' 符号。十进制常量以非零数字开头，由十进制数字序列组成。八进制常量由前缀 '0' 组成，后面可以选择性地跟只包含数字 '0' 到 '7' 的序列。十六进制常量由前缀 0x 或 0X 组成，后面跟十进制数字序列和字母 'a' (或 'A') 到 'f' (或 'F')，这些字母的值分别为 10 到 15。

如果 `base` 的值在 2 到 36 之间，主题序列的预期形式是表示整数的字母和数字序列，该整数的基数由 `base` 指定，可以选择性地在前面加 '+' 或 '-' 符号。从 'a' (或 'A') 到 'z' (或 'Z') 的字母被赋予 10 到 35 的值；只允许那些赋予值小于 `base` 的字母。如果 `base` 的值为 16，字符 0x 或 0X 可以选择性地在字母和数字序列之前，如果有符号则在符号之后。

主题序列被定义为输入字符串的最长子序列，从第一个非空白字节开始，且具有预期形式。如果输入字符串为空或完全由空白字节组成，或者如果第一个非空白字节不是符号或允许的字母或数字，则主题序列应不包含任何字符。

如果主题序列具有预期形式且 `base` 的值为 0，则从第一个数字开始的字符序列应被解释为整数常量。如果主题序列具有预期形式且 `base` 的值在 2 到 36 之间，它应用作转换的基数，将每个字母赋予上述值。如果主题序列以 <连字符> 开头，则结果值应为转换值的负值。如果 `endptr` 不是空指针，则指向最终字符串的指针应存储在 `endptr` 指向的对象中。

在 C [CX] 或 POSIX 语言环境之外，可以接受其他特定于语言环境的主题序列形式。

如果主题序列为空或不具有预期形式，则不执行转换；如果 `endptr` 不是空指针，`nptr` 的值应存储在 `endptr` 指向的对象中。

如果成功，这些函数不应改变 `errno` 的设置。

由于在错误时返回 0、`{LONG_MIN}` 或 `{LLONG_MIN}`、以及 `{LONG_MAX}` 或 `{LLONG_MAX}`，而这些值在成功时也是有效的返回值，因此希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strtol()` 或 `strtoll()`，然后检查 `errno`。

返回值

成功完成时，这些函数应返回转换后的值（如果有的话）。如果无法执行转换，应返回 0 [CX] 且 `errno` 可能被设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0 且 `errno` 应被设置为 [EINVAL]。

如果正确的值超出可表示值的范围，应返回 `{LONG_MIN}`、`{LONG_MAX}`、`{LLONG_MIN}` 或 `{LLONG_MAX}`（根据值的符号），并设置 `errno` 为 [ERANGE]。

错误

这些函数可能在以下情况下失败：

- [EINVAL] [CX] `base` 的值不受支持。
- [ERANGE] 要返回的值无法表示。

这些函数可能在以下情况下失败：

- [EINVAL] 无法执行转换。

示例

无。

应用程序用法

由于如果 `base` 的值不受支持, `*endptr` 的值是未指定的, 应用程序应在调用前确保 `base` 具有支持的值 (0 或 2 到 36 之间), 或者在检查 `*endptr` 之前检查 [EINVAL] 错误。

基本原理

无。

未来方向

无。

参见

- `fscanf()`
- `isalpha()`
- `strtod()`
- XBD `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

更新了描述以表明如果函数成功则不改变 `errno`。

Issue 6

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 在返回值和错误部分，如果无法执行转换，则添加 [EINVAL] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐进行了以下更改：

- 更新了 `strtol()` 原型。
- 添加了 `strtoll()` 函数。

Issue 7

应用了 POSIX.1-2008 技术勘误表 1 的 XSH/TC1-2008/0616 [453]、XSH/TC1-2008/0617 [105]、XSH/TC1-2008/0618 [453]、XSH/TC1-2008/0619 [453] 和 XSH/TC1-2008/0620 [453]。

应用了 POSIX.1-2008 技术勘误表 2 的 XSH/TC2-2008/0351 [892]、XSH/TC2-2008/0352 [584]、XSH/TC2-2008/0353 [796] 和 XSH/TC2-2008/0354 [892]。

Issue 8

应用了 Austin Group 缺陷 700，阐明了如何转换以 <连字符> 开头的主题序列。

应用了 Austin Group 缺陷 1163，阐明了输入字符串中空白字符的处理。

1.249. strtold

SYNOPSIS

```
#include <stdlib.h>

double strtod(const char *restrict nptr, char **restrict endptr)
float strtof(const char *restrict nptr, char **restrict endptr)
long double strtold(const char *restrict nptr, char **restrict
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 标准遵从 ISO C 标准。

这些函数应将由 `nptr` 指向的字符串的初始部分分别转换为 **double**、**float** 和 **long double** 表示形式。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 一个解释为浮点常量或表示无穷大或 NaN 的主题序列
3. 一个或多个无法识别的字符的最终字符串，包括输入字符串的终止 NUL 字符

然后它们应尝试将主题序列转换为浮点数，并返回结果。

主题序列的预期形式是一个可选的 '+' 或 '-' 符号，然后是以下形式之一：

- 一个非空的十进制数字序列，可选包含基数字符；然后是一个可选的指数部分，由字符 'e' 或字符 'E' 组成，可选后跟 '+' 或 '-' 字符，然后再后跟一个或多个十进制数字
- 一个 0x 或 0X，然后是一个非空的十六进制数字序列，可选包含基数字符；然后是一个可选的二进制指数部分，由字符 'p' 或字符 'P' 组成，可选后跟 '+' 或 '-' 字符，然后再后跟一个或多个十进制数字
- INF 或 INFINITY 之一，忽略大小写
- NAN 或 NAN(`n-char-sequence` *opt*) 之一，在 NAN 部分忽略大小写，其中：

```
n-char-sequence:
    digit
```

```
nondigit
n-char-sequence digit
n-char-sequence nondigit
```

主题序列定义为输入字符串的最长子序列，从第一个非空白字节开始，具有预期形式。如果输入字符串不具有预期形式，则主题序列不包含任何字符。

如果主题序列具有浮点数的预期形式，则从第一个数字或小数点字符（以先出现的为准）开始的字符序列应被解释为 C 语言的浮点常量，但应使用基数字符代替句点，并且如果在十进制浮点数中既没有指数部分也没有基数字符，或者在十六进制浮点数中没有二进制指数部分，则假定一个适当类型的值为零的指数部分跟在字符串中的最后一个数字之后。如果主题序列以 <连字符减号> 开头，则该序列应被解释为负数。字符序列 INF 或 INFINITY 应被解释为无穷大，如果在返回类型中可表示；否则应被解释为对于返回类型范围而言过大的浮点常量。字符序列 NAN 或 NAN(*n-char-sequence opt*) 应被解释为静态 NaN，如果在返回类型中支持；否则应被解释为不具有预期形式的主题序列部分； *n* - char 序列的含义由实现定义。指向最终字符串的指针存储在由 *endptr* 指向的对象中，前提是 *endptr* 不是空指针。

如果主题序列具有十六进制形式且 FLT_RADIX 是 2 的幂，则转换得到的值是正确舍入的。

[CX] 基数字符在当前区域设置（类别 LC_NUMERIC）中定义。在 POSIX 区域设置中，或在未定义基数字符的区域设置中，基数字符应默认为 <句点> ('.')。

除了 C [CX] 或 POSIX 区域设置之外，可以接受其他特定区域设置的主题序列形式。

如果主题序列为空或不具有预期形式，则不执行任何转换； *nptr* 的值存储在由 *endptr* 指向的对象中，前提是 *endptr* 不是空指针。

如果成功，这些函数不应改变 *errno* 的设置。

由于错误时返回 0，成功时也是有效的返回值，希望检查错误情况的应用程序应将 *errno* 设置为 0，然后调用 *strtod()*、*strtof()* 或 *strtold()*，然后检查 *errno*。

RETURN VALUE

成功完成后，这些函数应返回转换后的值。如果无法执行转换，应返回 0，并可能将 *errno* 设置为 [EINVAL]。

如果正确值会导致溢出且默认舍入生效，应返回 $\pm\text{HUGE_VAL}$ 、 $\pm\text{HUGE_VALF}$ 或 $\pm\text{HUGE_VALL}$ （根据值的符号），并将 *errno* 设置为 [ERANGE]。

如果正确值会导致下溢，应返回其大小不大于返回类型中最小规范化正数的值 [CX]，并将 `errno` 设置为 [ERANGE]。

ERRORS

这些函数在以下情况下应失败：

- **[ERANGE]**: 要返回的值会导致溢出且默认舍入生效 [CX]，或要返回的值会导致下溢。

这些函数在以下情况下可能失败：

- **[EINVAL]**: [CX] 无法执行转换。

以下章节为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

如果主题序列具有十六进制形式且 `FLT_RADIX` 不是 2 的幂，且结果不能精确表示，结果应是以十六进制浮点源值为相邻的适当内部格式中的两个数之一，额外规定误差对于当前舍入方向应具有正确的符号。

如果主题序列具有十进制形式且最多有 `DECIMAL_DIG`（在 `<float.h>` 中定义）个有效数字，结果应正确舍入。如果主题序列 `D` 具有十进制形式且有效数字超过 `DECIMAL_DIG`，考虑两个边界相邻十进制字符串 `L` 和 `U`，两者都具有 `DECIMAL_DIG` 个有效数字，使得 `L`、`D` 和 `U` 的值满足 `L <= D <= U`。结果应是通过根据当前舍入方向正确舍入 `L` 和 `U` 得到的（相等或相邻）值之一，额外规定相对于 `D` 的误差对于当前舍入方向应具有正确的符号。

ISO/IEC 9899:1999 标准对 `strtod()` 的引入可能改变符合 ISO/IEC 9899:1990 标准的格式良好应用程序的行为，因此也改变本标准的早期版本。一个这样的例子是：

```
int
what_kind_of_number (char *s)
{
```

```
char *endp;
double d;
long l;

d = strtod(s, &endp);
if (s != endp && *endp == '\0')
    printf("It's a float with value %g\n", d);
else
{
    l = strtol(s, &endp, 0);
    if (s != endp && *endp == '\0')
        printf("It's an integer with value %ld\n", l);
    else
        return 1;
}
return 0;
}
```

如果函数使用以下方式调用：

```
what_kind_of_number ("0x10")
```

符合 ISO/IEC 9899:1990 标准的库将导致函数打印：

```
It's an integer with value 16
```

使用 ISO/IEC 9899:1999 标准，结果是：

```
It's a float with value 16
```

行为的变化是由于包含十六进制表示法的浮点数，而不要求存在小数点或二进制指数。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `fscanf()`
- `isspace()`
- `localeconv()`
- `setlocale()`
- `strtol()`

XBD 7. Locale, `<float.h>`, `<stdlib.h>`

CHANGE HISTORY

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

更新 DESCRIPTION 以表明如果函数成功, `errno` 不会改变。

Issue 6

标记超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求源自与单一 UNIX 规范的对齐:

- 在 RETURN VALUE 和 ERRORS 部分中, 如果无法执行转换, 添加 [EINVAL] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐, 进行以下更改:

- 更新 `strtod()` 函数。
- 添加 `strtodf()` 和 `strtold()` 函数。
- 大幅修订 DESCRIPTION。

纳入 ISO/IEC 9899:1999 标准技术勘误 1。

应用 IEEE Std 1003.1-2001/Cor 1-2002, 项目 XSH/TC1/D6/61, 更正 RETURN VALUE 部分的第二段。此更改阐明了返回值的符号。

Issue 7

应用 Austin Group Interpretation 1003.1-2001 #015。

应用 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0610 [302], XSH/TC1-2008/0611 [94], 和 XSH/TC1-2008/0612 [105]。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0348 [584] 和 XSH/TC2-2008/0349 [796]。

Issue 8

应用 Austin Group Defect 1163, 阐明输入字符串中空白字符的处理。

应用 Austin Group Defect 1213, 更正 APPLICATION USAGE 部分中的一些印刷错误。

应用 Austin Group Defect 1302, 将这些函数与 ISO/IEC 9899:2018 标准对齐。

应用 Austin Group Defect 1686, 在 RETURN VALUE 部分的某些文本中添加 CX 底纹。

1.250. strtoll

SYNOPSIS

```
#include <stdlib.h>

long strtol(const char *restrict nptr, char **restrict endptr,
long long strtoll(const char *restrict nptr, char **restrict en
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

这些函数应将 `nptr` 指向的字符串的初始部分分别转换为 `long` 和 `long long` 类型的表示。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 一个解释为整数的主题序列，该整数以 `base` 值确定的某种基数表示
3. 一个或多个未识别字符的最终字符串，包括输入字符串的终止 NUL 字符。

然后它们应尝试将主题序列转换为整数，并返回结果。

如果 `base` 的值为 0，主题序列的预期形式为十进制常量、八进制常量或十六进制常量，其中任何一种都可能以 '+' 或 '-' 号开头。十进制常量以非零数字开头，由十进制数字序列组成。八进制常量由前缀 '0' 组成，可选地后跟仅由 '0' 到 '7' 的数字组成的序列。十六进制常量由前缀 0x 或 0X 组成，后跟十进制数字和字母 'a' (或 'A') 到 'f' (或 'F') 的序列，其值分别为 10 到 15。

如果 `base` 的值在 2 到 36 之间，主题序列的预期形式是表示以 `base` 指定的基数的整数的字母和数字序列，可选地以 '+' 或 '-' 号开头。从 'a' (或 'A') 到 'z' (或 'Z') 的字母被赋值为 10 到 35；只允许使用其赋值小于 `base` 值的字母。如果 `base` 的值为 16，字符 0x 或 0X 可以可选地位于字母和数字序列之前，如果存在符号，则在符号之后。

主题序列定义为输入字符串的最长子序列，从第一个非空白字节开始，具有预期形式。如果输入字符串为空或完全由空白字节组成，或者第一个非空白字节不是符号或允许的字母或数字，则主题序列应不包含任何字符。

如果主题序列具有预期形式且 `base` 的值为 0，则从第一个数字开始的字符序列应解释为整数常量。如果主题序列具有预期形式且 `base` 的值在 2 到 36 之

间，它应作为转换的基数，给每个字母赋予如上所述的值。如果主题序列以 <连字符减号> 开头，结果值应为转换值的负值。最终字符串的指针应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

在 C [CX] 或 POSIX 语言环境之外，可以接受其他特定于语言环境的主题序列形式。

如果主题序列为空或不具有预期形式，则不执行转换；`nptr` 的值应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果成功，这些函数不应更改 `errno` 的设置。

由于错误时返回 0、`{LONG_MIN}` 或 `{LLONG_MIN}` 和 `{LONG_MAX}` 或 `{LLONG_MAX}`，而成功时这些也是有效的返回值，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strtol()` 或 `strtoll()`，然后检查 `errno`。

RETURN VALUE

成功完成后，这些函数应返回转换后的值（如果有）。如果无法执行转换，应返回 0 [CX] 并可能将 `errno` 设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0 并将 `errno` 设置为 [EINVAL]。

如果正确的值超出可表示值的范围，应返回 `{LONG_MIN}`、`{LONG_MAX}`、`{LLONG_MIN}` 或 `{LLONG_MAX}`（根据值的符号），并将 `errno` 设置为 [ERANGE]。

ERRORS

这些函数应在以下情况下失败：

- [EINVAL] [CX] `base` 的值不受支持。
- [ERANGE] 要返回的值不可表示。

这些函数可能在以下情况下失败：

- [EINVAL] 无法执行转换。

EXAMPLES

无。

APPLICATION USAGE

由于如果 `base` 的值不受支持, `*endptr` 的值未指定, 应用程序应确保在调用前 `base` 具有受支持的值 (0 或在 2 到 36 之间), 或者在检查 `*endptr` 之前检查 [EINVAL] 错误。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- [fscanf\(\)](#)
- [isalpha\(\)](#)
- [strtod\(\)](#)
- XBD

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

DESCRIPTION 更新以指示如果函数成功则不更改 `errno`。

Issue 6

超出 ISO C 标准的扩展被标记。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐:

- 在 RETURN VALUE 和 ERRORS 部分, 如果无法执行转换, 则添加 [EINVAL] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐，进行以下更改：

- `strtol()` 原型更新。
- 添加 `strtoll()` 函数。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0616 [453], XSH/TC1-2008/0617 [105], XSH/TC1-2008/0618 [453], XSH/TC1-2008/0619 [453], 和 XSH/TC1-2008/0620 [453]。

应用了 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0351 [892], XSH/TC2-2008/0352 [584], XSH/TC2-2008/0353 [796], 和 XSH/TC2-2008/0354 [892]。

Issue 8

应用了 Austin Group Defect 700，阐明了如何转换以 <连字符减号> 开头的主题序列。

应用了 Austin Group Defect 1163，阐明了输入字符串中空白的处理。

1.251. strtoul

概要

```
#include <stdlib.h>

unsigned long strtoul(const char *restrict str,
                      char **restrict endptr, int base);
unsigned long long strtoull(const char *restrict str,
                           char **restrict endptr, int base);
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 卷遵循 ISO C 标准。

这些函数应将 `str` 指向的字符串的初始部分分别转换为 `unsigned long` 和 `unsigned long long` 类型的表示。首先，它们将输入字符串分解为三个部分：

1. 初始的、可能为空的空白字节序列
2. 解释为整数的主体序列，该整数的基数由 `base` 的值确定
3. 由一个或多个未识别字符组成的最终字符串，包括输入字符串的终止 NUL 字符

然后它们应尝试将主体序列转换为无符号整数，并返回结果。

如果 `base` 的值为 0，主体序列的预期形式为十进制常量、八进制常量或十六进制常量，其中任何一种前面都可以带有 '+' 或 '-' 符号。十进制常量以非零数字开头，由十进制数字序列组成。八进制常量由前缀 '0' 组成，后面可选地跟有仅由数字 '0' 到 '7' 组成的序列。十六进制常量由前缀 0x 或 0X 组成，后面跟有十进制数字和字母 'a' (或 'A') 到 'f' (或 'F') 的序列，这些字母分别表示值 10 到 15。

如果 `base` 的值在 2 到 36 之间，主体序列的预期形式是表示基数为 `base` 的整数的字母和数字序列，前面可选地带有 '+' 或 '-' 符号。从 'a' (或 'A') 到 'z' (或 'Z') 的字母被赋予值 10 到 35；只允许使用其赋值小于 `base` 的字母。

如果 `base` 的值为 16，字符 0x 或 0X 可以可选地位于字母和数字序列之前，如果有符号的话，位于符号之后。

主体序列定义为输入字符串的最长初始子序列，从第一个非空白字节开始，具有预期的形式。如果输入字符串为空或完全由空白字节组成，或者如果第一个非空白字节不是符号或允许的字母或数字，则主体序列应不包含任何字符。

如果主体序列具有预期形式且 `base` 的值为 0，从第一个数字开始的字符序列应被解释为整数常量。如果主体序列具有预期形式且 `base` 的值在 2 到 36 之间，它应用作转换的基数，为每个字母赋予如上所述的值。如果主体序列以 `0` 开头，结果值应为转换值的负数；此操作应在返回类型中执行。指向最终字符串的指针应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

在 C [CX] 或 POSIX 语言环境之外的其他语言环境中，可以接受其他特定于语言环境的主体序列形式。

如果主体序列为空或不具有预期形式，则不应执行转换；`str` 的值应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果成功，这些函数不应更改 `errno` 的设置。

由于在错误时返回 0、`{ULONG_MAX}` 和 `{ULLONG_MAX}`，这些值在成功时也是有效的返回值，因此希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strtoul()` 或 `strtoull()`，然后检查 `errno`。

返回值

成功完成后，这些函数应返回转换后的值（如果有）。如果无法执行转换，应返回 0 [CX]，并且 `errno` 可能设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0，`errno` 应设置为 [EINVAL]。

如果正确值超出了可表示值的范围，应返回 `{ULONG_MAX}` 或 `{ULLONG_MAX}`，`errno` 设置为 [ERANGE]。

错误

这些函数可能在以下情况下失败：

- [EINVAL]: [CX] `base` 的值不受支持。
- [ERANGE]: 要返回的值不可表示。

这些函数可能在以下情况下失败：

- [EINVAL]: [CX] 无法执行转换。

以下部分为补充信息。

示例

无。

应用程序用法

由于如果 `base` 的值不受支持, `*endptr` 的值是未指定的, 应用程序应在调用前确保 `base` 具有受支持的值 (0 或在 2 到 36 之间), 或者在检查 `*endptr` 之前检查 [EINVAL] 错误。

基本原理

无。

未来方向

无。

另请参阅

- `fscanf()`
- `isalpha()`
- `strtod()`
- `strtol()`

XBD `<stdlib.h>`

变更历史

首次发布于第 4 版

源自 ANSI C 标准。

第 5 版

更新了描述，指示如果函数成功，`errno` 不会更改。

第 6 版

超出了 ISO C 标准的扩展被标记。

以下对 POSIX 实现的新要求源于与单一 UNIX 规范的对齐：

- 为 `base` 的值不受支持的情况添加了 [EINVAL] 错误条件。
- 在返回值和错误部分，如果无法执行转换，添加了 [EINVAL] 可选错误条件。

为与 ISO/IEC 9899:1999 标准对齐进行了以下更改：

- 更新了 `strtoul()` 原型。
- 添加了 `strtoull()` 函数。

第 7 版

应用了 POSIX.1-2008、技术勘误 1、XSH/TC1-2008/0621 [105]、XSH/TC1-2008/0622 [453] 和 XSH/TC1-2008/0623 [453]。

应用了 POSIX.1-2008、技术勘误 2、XSH/TC2-2008/0355 [584] 和 XSH/TC2-2008/0356 [796]。

第 8 版

应用了 Austin Group 缺陷 700，阐明了如何转换以 `开头的主体序列。`

应用了 Austin Group 缺陷 1163，阐明了输入字符串中空白字符的处理。

1.252. strtoull — 将字符串转换为无符号长长整型

概要

```
#include <stdlib.h>

unsigned long long strtoull(const char *str, char **endptr, int
```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

这些函数应将 `str` 指向的字符串的初始部分分别转换为 `unsigned long` 和 `unsigned long long` 类型的表示。首先，它们将输入字符串分解为三个部分：

1. 一个初始的、可能为空的空白字节序列
2. 一个解释为主题序列的整数，该整数以由 `base` 值确定的某种基数表示
3. 一个由一个或多个无法识别的字符组成的最终字符串，包括输入字符串的终止 NUL 字符

然后它们应尝试将主题序列转换为无符号整数，并返回结果。

如果 `base` 的值为 0，主题序列的预期形式为十进制常量、八进制常量或十六进制常量，其中任何一种都可以前缀 '+' 或 '-' 符号。十进制常量以非零数字开头，由一系列十进制数字组成。八进制常量由前缀 '0' 组成，可选地后跟仅由 '0' 到 '7' 数字组成的序列。十六进制常量由前缀 0x 或 0X 组成，后跟十进制数字和字母 'a' (或 'A') 到 'f' (或 'F') 的序列，这些字母分别具有值 10 到 15。

如果 `base` 的值在 2 到 36 之间，主题序列的预期形式是表示由 `base` 指定基数的整数的字母和数字序列，可选地前缀 '+' 或 '-' 符号。从 'a' (或 'A') 到 'z' (或 'Z') 的字母被赋予值 10 到 35；只允许其赋值小于 `base` 值的字母。如果 `base` 的值为 16，字符 0x 或 0X 可选地可以在字母和数字序列之前，如果有符号的话跟在符号后面。

主题序列定义为输入字符串的最长初始子序列，从第一个非空白字节开始，具有预期形式。如果输入字符串为空或完全由空白字节组成，或者第一个非空白字节不是符号或允许的字母或数字，则主题序列应不包含任何字符。

如果主题序列具有预期形式且 `base` 的值为 0，从第一个数字开始的字符序列应被解释为整数常量。如果主题序列具有预期形式且 `base` 的值在 2 到 36 之间，它应被用作转换的基数，为每个字母赋予上述给定的值。如果主题序列以 < 连字符减号 > 开头，结果值应为转换值的负数；此操作应在返回类型中执行。指向最终字符串的指针应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

在 C [CX] 或 POSIX 语言环境之外，可以接受其他特定于语言环境的主题序列形式。

如果主题序列为空或不具有预期形式，则不执行转换；`str` 的值应存储在 `endptr` 指向的对象中，前提是 `endptr` 不是空指针。

如果成功，这些函数不应改变 `errno` 的设置。

由于错误时返回 0、`{ULONG_MAX}` 和 `{ULLONG_MAX}`，而这些值在成功时也是有效的返回值，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strtoul()` 或 `strtoull()`，然后检查 `errno`。

返回值

成功完成后，这些函数应返回转换后的值（如果有）。如果无法执行转换，应返回 0 [CX] 并可能将 `errno` 设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0 并将 `errno` 设置为 [EINVAL]。

如果正确值超出可表示值的范围，应返回 `{ULONG_MAX}` 或 `{ULLONG_MAX}` 并将 `errno` 设置为 [ERANGE]。

错误

`strtoull()` 函数应在以下情况失败：

- [EINVAL] — `base` 的值不受支持。
- [ERANGE] — 正确值超出可表示值的范围。

以下部分为参考信息。

示例

无。

应用程序用法

`strtoull()` 函数对于将无符号整数的字符串表示转换为其数值非常有用，具有适当的错误检查和各种进制数的处理。

原理

无。

未来方向

无。

另请参阅

- `strtoul()`
- `strtoimax()`
- `strtoumax()`
- `<stdlib.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 8

应用了 Austin Group Defect 1541，使功能与 ISO C 标准保持一致。

1.253. strtoumax

SYNOPSIS

```
#include <inttypes.h>

intmax_t strtointmax(const char *restrict nptr,
                      char **restrict endptr,
                      int base);
uintmax_t strtoumax(const char *restrict nptr,
                     char **restrict endptr,
                     int base);
```

DESCRIPTION

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

这些函数应等效于 `strtol()`、`strtoll()`、`strtoul()` 和 `strtoull()` 函数，不同之处在于字符串的初始部分应分别转换为 `intmax_t` 和 `uintmax_t` 表示形式。

RETURN VALUE

这些函数应返回转换后的值（如果有）。

如果无法执行转换，应返回零 [CX] 并可能将 `errno` 设置为 [EINVAL]。

[CX] 如果 `base` 的值不受支持，应返回 0 并将 `errno` 设置为 [EINVAL]。

如果正确的值超出可表示值的范围，应返回 {INTMAX_MAX}、{INTMAX_MIN} 或 {UINTMAX_MAX}（根据返回类型和值的符号，如果有），并将 `errno` 设置为 [ERANGE]。

ERRORS

这些函数在以下情况下应失败：

- [EINVAL] [CX] `base` 的值不受支持。
- [ERANGE] 要返回的值无法表示。

这些函数在以下情况下可能失败：

- [EINVAL] 无法执行转换。
-

以下部分为补充信息。

EXAMPLES

无。

APPLICATION USAGE

由于如果 `base` 的值不受支持, `*endptr` 的值是未指定的, 应用程序应在调用前确保 `base` 具有受支持的值 (0 或 2 到 36 之间), 或在检查 `*endptr` 前检查是否出现 [EINVAL] 错误。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- `strtol()`
- `strtoul()`
- XBD `<inttypes.h>`

CHANGE HISTORY

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

POSIX.1-2008、Technical Corrigendum 1、XSH/TC1-2008/0613 [453] 和 XSH/TC1-2008/0614 [453] 已应用。

1.254. `strxfrm`, `strxfrm_l` – 字符串转换

概要

```
#include <string.h>

size_t strxfrm(char *restrict s1, const char *restrict s2, size_t n);
/* [CX] 扩展 */
size_t strxfrm_l(char *restrict s1, const char *restrict s2,
                  size_t n, locale_t locale);
```

描述

对于 `strxfrm()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`strxfrm()` [CX] 和 `strxfrm_l()` 函数应转换由 `s2` 指向的字符串，并将结果字符串放置到由 `s1` 指向的数组中。转换的方式是：如果对两个转换后的字符串应用 `strcmp()`，它应返回大于、等于或小于 0 的值，分别对应于对相同的两个原始字符串 [CX] 在相同区域设置下应用 `strcoll()` [CX] 或 `strcoll_l()` 的结果。放置到由 `s1` 指向的结果数组中的字节数不超过 `n` 个，包括终止的 NUL 字符。如果 `n` 为 0，则允许 `s1` 为空指针。如果在重叠的对象之间进行复制，则行为未定义。

[CX] `strxfrm()` 和 `strxfrm_l()` 函数如果成功执行，不应更改 `errno` 的设置。

由于没有保留返回值来指示错误，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `strxfrm()` [CX] 或 `strxfrm_l()`，然后检查 `errno`。

[CX] 如果 `strxfrm_l()` 的 `locale` 参数是特殊的区域设置对象 `LC_GLOBAL_LOCALE` 或不是有效的区域设置对象句柄，则行为未定义。

返回值

成功完成后，`strxfrm()` [CX] 和 `strxfrm_l()` 应返回转换后字符串的长度（不包括终止的 NUL 字符）。如果返回值为 `n` 或更大，则由 `s1` 指向的数组

的内容未指定。

出错时, `strxfrm()` [CX] 和 `strxfrm_l()` 可能设置 `errno`, 但没有保留返回值来指示错误。

错误

这些函数可能失败的情况:

- [EINVAL] [CX] `s2` 参数指向的字符串包含排序序列域之外的字符。
-

以下部分为提供信息的部分。

示例

无。

应用程序用法

转换函数的设计使得两个转换后的字符串可以通过 `strcmp()` 进行排序, 其排序结果与当前区域设置 (类别 `LC_COLLATE`) 中的排序序列信息相对应。

当 `n` 为 0 时允许 `s1` 为空指针这一特性有助于在进行转换之前确定 `s1` 数组的大小。

基本原理

无。

未来方向

无。

另请参阅

- `strcmp()`

- `strcoll()`
- `<string.h>`

变更历史

首次发布于第 3 版。为与 ISO C 标准保持一致而包含在内。

第 5 版

更新了 DESCRIPTION 部分，以表明如果函数成功执行，`errno` 不会改变。

第 6 版

标记了超出 ISO C 标准的扩展。

以下对 POSIX 实现的新要求来源于与单一 UNIX 规范的对齐：

- 在 RETURN VALUE 和 ERRORS 部分中，如果无法执行转换，则添加了 [EINVAL] 可选错误条件。

`strxfrm()` 原型已更新，以与 ISO/IEC 9899:1999 标准对齐。

第 7 版

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `strxfrm_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0624 [283], XSH/TC1-2008/0625 [283]，和 XSH/TC1-2008/0626 [302]。

1.255. sysconf

概要 (SYNOPSIS)

```
#include <unistd.h>

long sysconf(int name);
```

描述 (DESCRIPTION)

`sysconf()` 函数为应用程序提供了一种方法来确定可配置系统限制或选项（变量）的当前值。实现必须支持下表中列出的所有变量，并且可以支持其他变量。

`name` 参数表示要查询的系统变量。下表列出了 `sysconf()` 可以返回的来自 `<limits.h>` 或 `<unistd.h>` 的最小系统变量集合，以及在 `<unistd.h>` 中定义的用于 `name` 的相应符号常量值。

系统变量 (System Variables)

变量	Name 参数值
{AIO_LISTIO_MAX}	_SC_AIO_LISTIO_MAX
{AIO_MAX}	_SC_AIO_MAX
{AIO_PRIO_DELTA_MAX}	_SC_AIO_PRIO_DELTA_MAX
{ARG_MAX}	_SC_ARG_MAX
{ATEXIT_MAX}	_SC_ATEXIT_MAX
{BC_BASE_MAX}	_SC_BC_BASE_MAX
{BC_DIM_MAX}	_SC_BC_DIM_MAX
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX

变量	Name 参数值
{BC_STRING_MAX}	_SC_BC_STRING_MAX
{CHILD_MAX}	_SC_CHILD_MAX
时钟滴答/秒	_SC_CLK_TCK
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
{DELAYTIMER_MAX}	_SC_DELAYTIMER_MAX
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
{HOST_NAME_MAX}	_SC_HOST_NAME_MAX
{IOV_MAX}	_SC_IOV_MAX
{LINE_MAX}	_SC_LINE_MAX
{LOGIN_NAME_MAX}	_SC_LOGIN_NAME_MAX
{NGROUPS_MAX}	_SC_NGROUPS_MAX
<code>getgrgid_r()</code> 和 <code>getgrnam_r()</code> 数据缓冲区的初始大小	_SC_GETGR_R_SIZE_MAX
<code>getpwuid_r()</code> 和 <code>getpwnam_r()</code> 数据缓冲区的初始大小	_SC_GETPW_R_SIZE_MAX
{MQ_OPEN_MAX}	_SC_MQ_OPEN_MAX
{MQ_PRIO_MAX}	_SC_MQ_PRIO_MAX
可用于运行线程的最大执行单元数量 +	_SC_NPROCESSORS_CONF
当前可用于运行线程的最大执行单元数量 +	_SC_NPROCESSORS_ONLN
支持的最高信号编号 +1	_SC_NSIG
{OPEN_MAX}	_SC_OPEN_MAX

变量	Name 参数值
{PAGE_SIZE}	_SC_PAGE_SIZE
{PAGESIZE}	_SC_PAGESIZE
{PTHREAD_DESTRUCTOR_ITERATIONS}	_SC_THREAD_DESTRUCTOR_ITERATIONS
{PTHREAD_KEYS_MAX}	_SC_THREAD_KEYS_MAX
{PTHREAD_STACK_MIN}	_SC_THREAD_STACK_MIN
{PTHREAD_THREADS_MAX}	_SC_THREAD_THREADS_MAX
{RE_DUP_MAX}	_SC_RE_DUP_MAX
{RTSIG_MAX}	_SC_RTSIG_MAX
{SEM_NSEMS_MAX}	_SC_SEM_NSEMS_MAX
{SEM_VALUE_MAX}	_SC_SEM_VALUE_MAX
{SIGQUEUE_MAX}	_SC_SIGQUEUE_MAX
{STREAM_MAX}	_SC_STREAM_MAX
{SYMLOOP_MAX}	_SC_SYMLOOP_MAX
{TIMER_MAX}	_SC_TIMER_MAX
{TTY_NAME_MAX}	_SC_TTY_NAME_MAX
{TZNAME_MAX}	_SC_TZNAME_MAX
_POSIX_ADVISORY_INFO	_SC_ADVISORY_INFO
_POSIX_BARRIERS	_SC_BARRIERS
_POSIX_ASYNCHRONOUS_IO	_SC_ASYNCHRONOUS_IO
_POSIX_CLOCK_SELECTION	_SC_CLOCK_SELECTION

变量	Name 参数值
_POSIX_CPUTIME	_SC_CPUTIME
_POSIX_DEVICE_CONTROL	_SC_DEVICE_CONTROL
_POSIX_FSYNC	_SC_FSYNC
_POSIX_IPV6	_SC_IPV6
_POSIX_JOB_CONTROL	_SC_JOB_CONTROL
_POSIX_MAPPED_FILES	_SC_MAPPED_FILES
_POSIX_MEMLOCK	_SC_MEMLOCK
_POSIX_MEMLOCK_RANGE	_SC_MEMLOCK_RANGE
_POSIX_MEMORY_PROTECTION	_SC_MEMORY_PROTECTION
_POSIX_MESSAGE_PASSING	_SC_MESSAGE_PASSING
_POSIX_MONOTONIC_CLOCK	_SC_MONOTONIC_CLOCK
_POSIX_PRIORITIZED_IO	_SC_PRIORITIZED_IO
_POSIX_PRIORITY_SCHEDULING	_SC_PRIORITY_SCHEDULING
_POSIX_RAW_SOCKETS	_SC_RAW_SOCKETS
_POSIX_READER_WRITER_LOCKS	_SC_READER_WRITER_LOCKS
_POSIX_REALTIME_SIGNALS	_SC_REALTIME_SIGNALS
_POSIX_REGEXP	_SC_REGEXP
_POSIX_SAVED_IDS	_SC_SAVED_IDS
_POSIX_SEMAPHORES	_SC_SEMAPHORES
_POSIX_SHARED_MEMORY_OBJECTS	_SC_SHARED_MEMORY_OBJECTS

变量	Name 参数值
_POSIX_SHELL	_SC_SHELL
_POSIX_SPAWN	_SC_SPAWN
_POSIX_SPIN_LOCKS	_SC_SPIN_LOCKS
_POSIX_SPORADIC_SERVER	_SC_SPORADIC_SERVER
_POSIX_SS_REPL_MAX	_SC_SS_REPL_MAX
_POSIX_SYNCHRONIZED_IO	_SC_SYNCHRONIZED_IO
_POSIX_THREAD_ATTR_STACKADDR	_SC_THREAD_ATTR_STACKADDR
_POSIX_THREAD_ATTR_STACKSIZE	_SC_THREAD_ATTR_STACKSIZE
_POSIX_THREAD_CPUTIME	_SC_THREAD_CPUTIME
_POSIX_THREAD_PRIO_INHERIT	_SC_THREAD_PRIO_INHERIT
_POSIX_THREAD_PRIO_PROTECT	_SC_THREAD_PRIO_PROTECT
_POSIX_THREAD_PRIORITY_SCHEDULING	_SC_THREAD_PRIORITY_SCHEDULING
_POSIX_THREAD_PROCESS_SHARED	_SC_THREAD_PROCESS_SHARED
_POSIX_THREAD_ROBUST_PRIO_INHERIT	_SC_THREAD_ROBUST_PRIO_INHERIT
_POSIX_THREAD_ROBUST_PRIO_PROTECT	_SC_THREAD_ROBUST_PRIO_PROTECT
_POSIX_THREAD_SAFE_FUNCTIONS	_SC_THREAD_SAFE_FUNCTIONS
_POSIX_THREAD_SPORADIC_SERVER	_SC_THREAD_SPORADIC_SERVER
_POSIX_THREADS	_SC_THREADS
_POSIX_TIMEOUTS	_SC_TIMEOUTS
_POSIX_TIMERS	_SC_TIMERS

变量	Name 参数值
_POSIX_TYPED_MEMORY_OBJECTS	_SC_TYPED_MEMORY_OBJECTS
_POSIX_VERSION	_SC_VERSION
_POSIX_V8_ILP32_OFF32	_SC_V8_ILP32_OFF32
_POSIX_V8_ILP32_OFFBIG	_SC_V8_ILP32_OFFBIG
_POSIX_V8_LP64_OFF64	_SC_V8_LP64_OFF64
_POSIX_V8_LPBIG_OFFBIG	_SC_V8_LPBIG_OFFBIG
_POSIX_V7_ILP32_OFF32	_SC_V7_ILP32_OFF32
_POSIX_V7_ILP32_OFFBIG	_SC_V7_ILP32_OFFBIG
_POSIX_V7_LP64_OFF64	_SC_V7_LP64_OFF64
_POSIX_V7_LPBIG_OFFBIG	_SC_V7_LPBIG_OFFBIG
_POSIX2_C_BIND	_SC_2_C_BIND
_POSIX2_C_DEV	_SC_2_C_DEV
_POSIX2_CHAR_TERM	_SC_2_CHAR_TERM
_POSIX2_FORT_RUN	_SC_2_FORT_RUN
_POSIX2_LOCALEDEF	_SC_2_LOCALEDEF
_POSIX2_SW_DEV	_SC_2_SW_DEV
_POSIX2_UPE	_SC_2_UPE
_POSIX2_VERSION	_SC_2_VERSION
_XOPEN_CRYPT	_SC_XOPEN_CRYPT
_XOPEN_ENH_I18N	_SC_XOPEN_ENH_I18N

变量	Name 参数值
_XOPEN_REALTIME	_SC_XOPEN_REALTIME
_XOPEN_REALTIME_THREADS	_SC_XOPEN_REALTIME_THREADS
_XOPEN_SHM	_SC_XOPEN_SHM
_XOPEN_UNIX	_SC_XOPEN_UNIX
_XOPEN_UUCP	_SC_XOPEN_UUCP
_XOPEN_VERSION	_SC_XOPEN_VERSION

† 执行单元的性质以及执行单元被视为可用、可以变得可用的精确条件，或者它可以并行执行多少个线程，都是由实现定义的。

返回值 (RETURN VALUE)

如果 `name` 是无效值，`sysconf()` 应返回 -1 并设置 `errno` 来指示错误。如果与 `name` 对应的变量在 `<limits.h>` 中被描述为最大值或最小值但该变量没有限制，`sysconf()` 应返回 -1 而不改变 `errno` 的值。注意，不确定的限制并不意味着无限限制；参见 `<limits.h>`。

否则，`sysconf()` 应返回系统上该变量的当前值。返回的值不应比应用程序使用实现的 `<limits.h>` 或 `<unistd.h>` 编译时描述给应用程序的相应值更具限制性。除了 `_SC_NPROCESSORS_ONLN` 之外，为 `name` 参数返回的值在调用进程的生命周期内不应改变，但是 `sysconf(_SC_OPEN_MAX)` 在调用 `setrlimit()` 改变 RLIMIT_NOFILE 软限制之前和之后可能返回不同的值。

如果与 `name` 对应的变量依赖于不支持的选项，则结果未指定。

错误 (ERRORS)

`sysconf()` 函数在以下情况下应失败：

- [EINVAL] - `name` 参数的值无效。

应用程序用法 (APPLICATION USAGE)

由于 -1 在成功情况下也是允许的返回值，希望检查错误情况的应用程序应将 `errno` 设置为 0，然后调用 `sysconf()`，如果返回 -1，则检查 `errno` 是否为非零值。

应用程序开发人员应在获取和使用相关变量值（如 `_POSIX_SS_REPL_MAX`）之前检查某个选项（如 `_POSIX_SPORADIC_SERVER`）是否受支持。

虽然 `_SC_NPROCESSORS_CONF` 和 `_SC_NPROCESSORS_ONLN` 查询为一类"高负载"应用程序提供了一种估计可创建的线程最优数量以最大化吞吐量的方法，但现实世界环境存在影响可实现的实际效率的复杂情况。例如：

- 系统上可能运行着多个"高负载"应用程序。
- 系统可能在使用电池电源，应用程序应与系统协调以确保长时间运行的任务可以暂停、恢复并在断电情况下也能成功完成。

如果一个可移植的"高负载"应用程序想要避免使用扩展，其开发人员可能希望基于长时间运行任务的逻辑分区来创建线程，或者利用启发式方法（如 CPU 时间与实时时间的比率）。

原理 (RATIONALE)

此功能是为了满足应用程序开发人员和处理许多国际系统配置的系统供应商的要求而添加的。它与 `pathconf()` 和 `fpathconf()` 密切相关。

虽然符合规范的应用程序可以通过从不要求超过 POSIX.1-2024 卷中发布的最小值资源来在所有系统上运行，但该应用程序能够使用任何给定系统上可用资源数量的实际值是有用的。为此，应用程序使用 `<limits.h>` 或 `<unistd.h>` 中符号常量的值。

然而，一旦编译完成，应用程序必须仍然能够处理可用资源量增加的情况。为此，应用程序可能需要一种在执行时确定资源数量或选项存在性的方法。

提供两个示例：

1. 应用程序可能希望在有作业控制和无作业控制的系统上表现不同。希望向计算机架构的所有实例只分发单一二进制包的应用程序供应商，如果只依赖 POSIX.1-2024 卷中发布的 `<unistd.h>` 值，将被迫假设作业控制永远不可用。
2. 国际应用程序供应商有时需要知道每秒时钟滴答数的知识。没有这些功能，他们将需要要么以源代码形式部分分发其应用程序，要么为其运营的各个国家提供 50 Hz 和 60 Hz 版本。

正是许多应用程序实际上以可执行形式广泛分发的认识导致了这个功能的产生。如果仅限于头文件中最具限制性的值，此类应用程序将必须准备接受最小微计算机提供的环境限制。虽然这完全是可移植的，但人们普遍认为它们应该能够利用大型系统提供的功能，而不受源代码和目标代码分发相关的限制。

在讨论此功能期间，有人指出应用程序几乎总是可以通过适当测试各种函数本身来在运行时识别某个值可能是什么。而且，无论如何，它总是可以被编写为充分处理各种函数的错误返回。最终，人们认为这给应用程序开发人员施加了不合理程度的复杂性和成熟度要求。

这个运行时功能并不意味着提供应用程序必须多次检查的不断变化的值。这些值被认为变化的频率不超过每个系统初始化一次，例如由系统管理员或操作员使用自动配置程序进行的初始化。POSIX.1-2024 卷规定它们在进程生命周期内不应改变。

一些值适用于整个系统，而另一些值在文件系统或目录级别变化。后者在 [fpathconf\(\)](#) 中描述。

注意，所有返回的值必须能够表示为整数。字符串值曾被考虑，但由于其实现和使用的额外复杂性而被拒绝。

某些值（如 {PATH_MAX}）有时太大，不能用于分配数组等操作。[sysconf\(\)](#) 函数返回负值以表明在这种情况下甚至没有定义这个符号常量。

与 [pathconf\(\)](#) 类似，这允许实现没有限制。当某个资源无限时，返回表示已达到其他某个资源限制的错误是符合要求的行为。

未来方向 (FUTURE DIRECTIONS)

无。

参见 (SEE ALSO)

- [confstr\(\)](#)
- [fpathconf\(\)](#)
- [<limits.h>](#)
- [<unistd.h>](#)
- [getconf](#)

1.256. time — 获取时间

SYNOPSIS

```
#include <time.h>

time_t time(time_t *tloc);
```

DESCRIPTION

[CX] 本参考页上描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`time()` 函数应返回自纪元 (Epoch) 以来经过的秒数时间值 [CX]。

`tloc` 参数指向一个区域，该区域也存储返回值。如果 `tloc` 是空指针，则不存储任何值。

RETURN VALUE

成功完成后，`time()` 应返回时间值。否则，应返回 `(time_t)-1`。

ERRORS

`time()` 函数可能在以下情况下失败：

- **[EOVERFLOW]** [CX] 自纪元以来的秒数无法放入 `time_t` 类型的对象中。

以下章节为信息性内容。

EXAMPLES

获取当前时间

以下示例使用 `time()` 函数计算自纪元以来经过的秒数，使用 `localtime()` 将该值转换为分解时间，并使用 `asctime()` 将分解时间值转换为可打印字符串。

```
#include <stdio.h>
#include <time.h>

int main(void)
{
    time_t result;

    result = time(NULL);
    printf("%s%ju secs since the Epoch\n",
           asctime(localtime(&result)),
           (uintmax_t)result);
    return(0);
}
```

此示例以类似以下格式将当前时间写入 `stdout`：

```
Wed Jun 26 10:32:15 1996
835810335 secs since the Epoch
```

为事件计时

以下示例获取当前时间，以用户格式输出，并打印到正在计时的事件的分钟数。

```
#include <time.h>
#include <stdio.h>
...
time_t now;
int minutes_to_event;
...
time(&now);
minutes_to_event = ...;
printf("The time is ");
puts(asctime(localtime(&now)));
printf("There are %d minutes to the event.\n",
       minutes_to_event);
...
```

APPLICATION USAGE

无。

RATIONALE

`time()` 函数以秒为单位返回值，而 `clock_gettime()` 返回 `struct timespec` (秒和纳秒)，因此能够返回更精确的时间。`times()` 函数也比 `time()` 能够提供更高的精度，因为它以时钟滴答为单位返回值，尽管它返回的是从任意点 (如系统启动时间) 开始经过的时间，而不是从纪元开始。

本标准的早期版本允许 `time_t` 的宽度小于 64 位。32 位有符号整数 (在许多历史实现中使用) 在 2038 年会失效，虽然 32 位无符号整数要到 2106 年才会失效，但首选的解决方案是使 `time_t` 更宽，而不是使其变为无符号类型。

在某些系统上，`time()` 函数使用系统调用实现，该系统调用除了返回值之外不返回错误条件。在这些系统上，无法区分有效和无效的返回值，因此无法可靠地检测溢出条件。

使用 `<time.h>` 头文件而不是 `<sys/types.h>` 允许与 ISO C 标准兼容。

许多历史实现 (包括第 7 版) 和 1984 年的 /usr/group 标准使用 `long` 而不是 `time_t`。为了与 ISO C 标准保持一致，POSIX.1-2024 本卷使用后一种类型。

FUTURE DIRECTIONS

无。

SEE ALSO

- `asctime()`
- `clock()`
- `clock_getres()`
- `ctime()`
- `difftime()`
- `futimens()`
- `gmtime()`
- `localtime()`
- `mktime()`
- `strftime()`
- `strptime()`

- `times()`

XBD `<time.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

添加了 EXAMPLES、RATIONALE 和 FUTURE DIRECTIONS 章节。

Issue 7

应用了 POSIX.1-2008， Technical Corrigendum 1， XSH/TC1-2008/0663 [106]、 XSH/TC1-2008/0664 [350]、 XSH/TC1-2008/0665 [106]、 XSH/TC1-2008/0666 [350] 和 XSH/TC1-2008/0667 [350]。

Issue 8

应用了 Austin Group Defect 1330，移除了过时的接口。

应用了 Austin Group Defect 1462，更改了 RATIONALE 和 FUTURE DIRECTIONS 章节。

信息性文本结束。

1.257. timer_create - 创建每进程定时器

SYNOPSIS (概要)

```
#include <signal.h>
#include <time.h>

int timer_create(clockid_t clockid,
                 struct sigevent *restrict evp,
                 timer_t *restrict timerid);
```

DESCRIPTION (描述)

`timer_create()` 函数应使用指定的时钟 `clock_id` 作为计时基础来创建一个每进程定时器。`timer_create()` 函数应在 `timerid` 引用的位置返回一个类型为 `timer_t` 的定时器 ID，用于在定时器请求中标识该定时器。在定时器被删除之前，此定时器 ID 在调用进程内应是唯一的。特定的时钟 `clock_id` 在 `<time.h>` 中定义。返回的 ID 所对应的定时器在从 `timer_create()` 返回时应处于解除武装状态。

如果 `evp` 参数非 NULL，则指向一个 `sigevent` 结构。这个由应用程序分配的结构定义了定时器到期时按照信号生成和传递规范发生的异步通知。如果 `evp` 参数为 NULL，则效果相当于 `evp` 参数指向一个 `sigevent` 结构，其中 `sigev_notify` 成员具有值 `SIGEV_SIGNAL`，`sigev_signo` 具有默认信号编号，`sigev_value` 成员具有定时器 ID 的值。

每个实现应定义一组可用作每进程定时器计时基础的时钟。所有实现都应支持 `CLOCK_REALTIME` 和 `CLOCK_MONOTONIC` 作为 `clock_id` 的值。

每进程定时器不应通过 `fork()` 由子进程继承，并应通过 `exec` 被解除武装和删除。

如果定义了 `_POSIX_CPUTIME`，实现应支持表示调用进程 CPU 时间时钟的 `clock_id` 值。

如果定义了 `_POSIX_THREAD_CPUTIME`，实现应支持表示调用线程 CPU 时间时钟的 `clock_id` 值。

如果 `clock_id` 定义的值对应于不同于调用函数的进程或线程的进程或线程的 CPU 时间时钟，`timer_create()` 函数是否会成功是由实现定义的。

如果 `evp->sigev_notify` 是 `SIGEV_THREAD` 且 `sev->sigev_notify_attributes` 不为 `NULL`，如果 `sev->sigev_notify_attributes` 指向的属性具有通过调用 `pthread_attr_setstack()` 指定的线程栈地址，则如果信号生成多次，结果未指定。

RETURN VALUE (返回值)

如果调用成功，`timer_create()` 应返回零，并将 `timerid` 引用的位置更新为一个 `timer_t`，该值可以传递给每进程定时器调用。如果发生错误，函数应返回值 `-1` 并设置 `errno` 以指示错误。如果发生错误，`timerid` 的值未定义。

ERRORS (错误)

`timer_create()` 函数应在以下情况下失败：

- **[EAGAIN]**
 - 系统缺乏足够的信号排队资源来满足请求。
 - 调用进程已经创建了此实现允许的所有定时器。
- **[EINVAL]**
 - 指定的时钟 ID 未定义。
- **[ENOTSUP]**
 - 实现不支持创建附加到由 `clock_id` 指定的 CPU 时间时钟的定时器，且该时钟与不同于调用 `timer_create()` 的进程或线程的进程或线程相关联。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

如果创建的定时器将 `evp->sigev_notify` 设置为 `SIGEV_THREAD`，并且 `evp->sigev_notify_attributes` 指向的属性具有通过调用

`pthread_attr_setstack()` 指定的线程栈地址，则专用于线程栈的内存无法恢复。原因是响应定时器到期而创建的线程是以分离方式创建的，或者如果线程属性的 `detachstate` 是 `PTHREAD_CREATE_JOINABLE`，则以未指定的方式创建。在这两种情况下，调用 `pthread_join()` 都是无效的，这导致无法确定创建的线程的生命周期，从而意味着栈内存无法重用。

RATIONALE (原理)

周期性定时器溢出和资源分配

指定的定时器设施可能在此选项支持的实现上传递实时信号（即排队信号）。由于实时应用程序不能承受丢失异步事件（如定时器到期或异步 I/O 完成）的通知，因此必须能够确保在事件发生时有足够的资源来传递信号。通常，这不是问题，因为请求与后续信号生成之间存在一一对应的关系。如果请求无法分配信号传递资源，它可以通过 [EAGAIN] 错误使调用失败。

周期性定时器是特例。单个请求可以生成未指定数量的信号。如果请求进程能以生成信号的速度服务信号，从而使信号传递资源可用于传递后续周期性定时器到期信号，这就不是问题。但通常，这无法保证——周期性定时器信号的处理可能“溢出”；即后续周期性定时器到期可能在当前待处理信号被传递之前发生。

另外，对于信号，根据 POSIX.1-1990 标准，如果生成了待处理信号的后续出现，是否为每次出现都传递一个信号是由实现定义的。这对某些实时应用程序来说是不够的。因此需要一种机制来允许应用程序检测有多少定时器到期被延迟，而不需要无限量的系统资源来存储延迟的到期。

指定的设施提供了溢出计数。溢出计数定义为在定时器到期信号生成时间和信号传递时间之间发生的额外定时器到期数量。如果信号捕获函数关心溢出，可以在入口时检索此计数。使用此方法，周期性定时器只需要一个“信号排队资源”，该资源可以在调用 `timer_create()` 函数时分配。

定义了一个函数来检索溢出计数，以便应用程序不需要分配静态存储来包含计数，实现也不需要在定时器到期时异步更新此存储。但对于某些高频周期性应用程序，每次定时器到期时的额外系统调用开销可能过于昂贵。如定义的函数允许实现在用户空间中维护与 `timerid` 相关联的溢出计数。
`timer_getoverrun()` 函数然后可以实现为使用 `timerid` 参数（可能只是指向包含计数器的用户空间结构的指针）来定位溢出计数的宏，没有系统调用开销。其他不太关心此类应用程序的实现可以通过在系统结构中维护计数来避免用户空间的异步更新，代价是额外的系统调用来获取计数。

定时器到期信号参数

实时信号功能支持传递给扩展信号处理器的应用程序特定数据。此值由应用程序与要传递的信号编号一起在 `sigevent` 结构中明确指定。应用程序定义的值的类型可以是整数常量或指针。这种值的显式规范，而不是总是发送定时器 ID，是基于现有实践选择的。

实时应用程序（在非 POSIX 系统或实时扩展 POSIX 系统上）通常使用事件处理器的参数作为 `switch` 语句的 `case` 标签或指向应用程序定义数据结构的指针。由于 `timer_id` 由 `timer_create()` 函数动态分配，它们不能用于这些功能而不在信号处理器中产生额外的应用程序开销；例如，搜索保存的定时器 ID 数组以将 ID 与常量或应用程序数据结构关联。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (另请参阅)

- `clock_getres()`
- `timer_delete()`
- `timer_getoverrun()`
- `<signal.h>`
- `<time.h>`

CHANGE HISTORY (变更历史)

首次在 Issue 5 中发布。包含用于与 POSIX 实时扩展对齐。

Issue 6

`timer_create()` 函数被标记为 Timers 选项的一部分。

[ENOSYS] 错误条件已被删除，因为如果实现不支持 Timers 选项，不需要提供存根。

添加了 CPU 时间时钟以与 IEEE Std 1003.1d-1999 对齐。

DESCRIPTION 部分已更新以与 IEEE Std 1003.1j-2000 对齐，在 Monotonic Clock 选项下添加了对 CLOCK_MONOTONIC 时钟的要求。

`restrict` 关键字已添加到 `timer_create()` 原型以与 ISO/IEC 9899:1999 标准对齐。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/138，更新了 DESCRIPTION 和 APPLICATION USAGE 部分以描述通知方法设置为 SIGEV_THREAD 时创建定时器的情况。

Issue 7

`timer_create()` 函数已从 Timers 选项移至 Base。

Issue 8

应用了 Austin Group 缺陷 1116，删除了对本标准早期版本中存在的 Realtime Signals Extension 选项的引用。

应用了 Austin Group 缺陷 1346，要求支持 Monotonic Clock。

1.258. timer_delete

名称

```
#include <time.h>

int timer_delete(timer_t timerid);
```

描述

`timer_delete()` 函数删除指定的定时器 `timerid`，该定时器先前由 `timer_create()` 函数创建。如果在调用 `timer_delete()` 时定时器已启动，则其行为应等同于在删除前自动解除定时器的启动状态。对于已删除定时器的待处理信号的处理方式是未指定的。

如果传递给 `timer_delete()` 的 `timerid` 参数不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器ID，则其行为是未定义的。

返回值

如果成功，`timer_delete()` 函数应返回零值。否则，函数应返回 -1 值并设置 `errno` 来指示错误。

错误

未定义任何错误。

以下部分为参考资料。

示例

无。

应用程序用法

无。

原理

如果实现检测到传递给 `timer_delete()` 的 `timerid` 参数值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器ID，建议函数应失败并报告 [EINVAL] 错误。

未来方向

无。

参见

- [timer_create\(\)](#)
- [<time.h>](#)

变更历史

首次发布于 Issue 5。包含在内以与 POSIX 实时扩展对齐。

Issue 6

[timer_delete\(\)](#) 函数被标记为定时器选项的一部分。

如果实现不支持定时器选项，则无需提供存根函数，因此移除了 [ENOSYS] 错误条件。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/139，更新了错误部分，使 [EINVAL] 错误变为可选的。

Issue 7

[timer_delete\(\)](#) 函数从定时器选项移动到基础部分。

应用了 POSIX.1-2008，技术勘误表 2，XSH/TC2-2008/0369 [659]。

参考资料结束。

版权所有 © 2001-2024 IEEE 和 The Open Group，保留所有权利

1.259. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器

概要

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

描述

`timer_gettime()` 函数应将指定定时器 `timerid` 到期之前的时间量和定时器的重载值存储到 `value` 参数指向的空间中。该结构的 `it_value` 成员应包含定时器到期之前的时间量，如果定时器已解除武装则为零。即使定时器是以绝对时间武装的，该值也作为定时器到期前的间隔返回。`value` 的 `it_interval` 成员应包含由 `timer_settime()` 最后设置的重载值。

`timer_settime()` 函数应从 `value` 参数的 `it_value` 成员设置由 `timerid` 指定的定时器的下次到期时间，如果 `value` 的 `it_value` 成员非零，则武装定时器。如果在调用 `timer_settime()` 时指定的定时器已经武装，此调用应将下次到期的时间重置为指定的 `value`。如果 `value` 的 `it_value` 成员为零，则定时器应被解除武装。解除武装或重置具有挂起到期通知的定时器的效果是未指定的。

如果标志 `TIMER_ABSTIME` 未在参数 `flags` 中设置，`timer_settime()` 的行为应如同下次到期的时间被设置为等于 `value` 的 `it_value` 成员指定的间隔。也就是说，定时器应在调用发出后的 `it_value` 纳秒后到期。如果标志 `TIMER_ABSTIME` 在参数 `flags` 中设置，`timer_settime()` 的行为应如同下次到期的时间被设置为等于 `value` 的 `it_value` 成员指定的绝对时间与 `timerid` 关联的时钟当前值之间的差值。也就是说，当时钟达到 `value` 的 `it_value` 成员指定的值时，定时器应到期。如果指定的时间已经过去，函数应成功并发出到期通知。

定时器的重载值应设置为 `value` 的 `it_interval` 成员指定的值。当定时器以非零的 `it_interval` 武装时，指定了周期性（或重复性）定时器。

介于指定定时器分辨率的两个连续非负整数倍之间的时间值应向上舍入到较大的分辨率倍数。量化误差不应导致定时器在舍入时间值之前到期。

如果参数 `ovalue` 不为 `NULL`, `timer_settime()` 函数应将表示定时器原来到期前时间量的值 (如果定时器已解除武装则为零) 以及原来的定时器重载值一起存储在 `ovalue` 引用的位置。定时器不应在预定时间之前到期。

在任何时间点, 对于给定的定时器, 只有一个信号会被排队到进程。当一个信号仍在挂起中的定时器到期时, 不会排队信号, 并且会发生定时器超限。当定时器到期信号传递给或被进程接受时, `timer_getoverrun()` 函数应返回指定定时器的定时器到期超限计数。返回的超限计数包含在信号生成 (排队) 时间和传递或接受时间之间发生的额外定时器到期次数, 最多但不包括实现定义的最大值 `{DELAYTIMER_MAX}`。如果此类额外到期次数大于或等于 `{DELAYTIMER_MAX}`, 则超限计数应设置为 `{DELAYTIMER_MAX}`。`timer_getoverrun()` 返回的值适用于定时器最近的到期信号传递或接受。如果没有为定时器传递到期信号, `timer_getoverrun()` 的返回值是未指定的。

如果传递给 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID, 则行为是未定义的。

返回值

如果 `timer_getoverrun()` 函数成功, 它应返回定时器到期超限计数, 如上所述。

如果 `timer_gettime()` 或 `timer_settime()` 函数成功, 应返回值 0。

如果任何这些函数发生错误, 应返回值 -1, 并设置 `errno` 以指示错误。

错误

`timer_settime()` 函数在以下情况下应失败:

- **[EINVAL]**

`value` 结构指定了小于零或大于等于 10 亿的纳秒值, 并且该结构的 `it_value` 成员没有指定零秒和零纳秒。

`timer_settime()` 函数在以下情况下可能失败:

- **[EINVAL]**

`value` 的 `it_interval` 成员不为零, 并且定时器是通过创建新线程

(`sigev_sigev_notify` 为 `SIGEV_THREAD`) 创建通知的，并且在 `sigev_notify_attributes` 指向的线程属性中设置了固定堆栈地址。

应用程序用法

当定时器到期通过创建新线程来发信号时，使用固定堆栈地址是有问题的。由于不能假定为一个到期创建的线程在定时器的下次到期之前已经完成，因此可能发生两个线程同时使用相同内存作为堆栈的情况。这是无效的，并会产生未定义的结果。

基本原理

实际的时钟以有限速率滴答，100 赫兹和 1000 赫兹是常见的速率。此滴答速率的倒数是时钟分辨率，也称为时钟粒度，无论哪种情况都表示为时间持续时间，对于这些常见速率分别为 10 毫秒和 1 毫秒。实际时钟的粒度意味着，如果快速连续两次读取给定时钟，可能两次获得相同的时间值；并且定时器必须等待理论到期时间之后的下一个时钟滴答，以确保定时器永远不会提前返回。还要注意，时钟的粒度可能比用于设置和获取时间及间隔值的数据格式的分辨率明显粗糙。还要注意，一些实现可能选择调整时间和/或间隔值以精确匹配底层时钟的滴答。

POSIX.1-2024 本卷定义了允许应用程序确定实现支持的时钟分辨率的函数，并要求实现文档化定时器和 `nanosleep()` 支持的分辨率（如果它们与支持的时钟分辨率不同）。这更多的是采购问题，而不是运行时应用程序问题。

如果实现检测到传递给 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID，建议函数应失败并报告 `[EINVAL]` 错误。

未来方向

无。

另请参阅

- `clock_getres()`
- `timer_create()`

- `<time.h>`

更改历史

首次在 Issue 5 中发布。包含用于与 POSIX 实时扩展对齐。

Issue 6

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数被标记为定时器选项的一部分。

[ENOSYS] 错误条件已被删除，因为如果实现不支持定时器选项，则不需要提供存根。

[EINVAL] 错误条件更新为包括以下内容：“并且该结构的 `it_value` 成员没有指定零秒和零纳秒。”此更改是为了 IEEE PASC 解释 1003.1 #89。

`timer_getoverrun()` 的描述更新为阐明：“如果没有为定时器传递到期信号，或者不支持实时信号扩展，`timer_getoverrun()` 的返回值是未指定的”。

为了与 ISO/IEC 9899:1999 标准对齐，`restrict` 关键字添加到 `timer_settime()` 原型中。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/140，更新错误部分，使强制性的 [EINVAL] 错误（“`timerid` 参数不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的 ID”）变为可选的。

应用了 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/141，更新错误部分以包括当定时器使用设置为 SIGEV_THREAD 的通知方法创建时的可选 [EINVAL] 错误。还添加了应用程序用法文本。

Issue 7

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数从定时器选项移动到基础。

与实时信号扩展选项相关的功能移动到基础。

应用了 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0370 [659]。

1.260. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器

SYNOPSIS 概要

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

DESCRIPTION 描述

`timer_gettime()` 函数应将指定定时器 `timerid` 到期前的剩余时间量和定时器的重载值存储到 `value` 参数指向的空间中。此结构的 `it_value` 成员应包含定时器到期前的时间量，如果定时器已解除武装则为零。即使定时器是用绝对时间武装的，此值也作为定时器到期前的间隔返回。`value` 的 `it_interval` 成员应包含由 `timer_settime()` 最后设置的重载值。

`timer_settime()` 函数应从 `value` 参数的 `it_value` 成员设置指定定时器 `timerid` 到下一次到期的时间，如果 `value` 的 `it_value` 成员非零，则武装该定时器。如果在调用 `timer_settime()` 时指定的定时器已经被武装，此调用应重置到下一次到期的时间为指定的 `value`。如果 `value` 的 `it_value` 成员为零，则定时器应被解除武装。对于具有挂起到期通知的定时器，解除武装或重置的效果是未指定的。

如果参数 `flags` 中未设置标志 `TIMER_ABSTIME`，`timer_settime()` 应表现为到下一次到期的时间设置为等于 `value` 的 `it_value` 成员指定的间隔。即，定时器应在调用发生后的 `it_value` 纳秒到期。如果参数 `flags` 中设置了标志 `TIMER_ABSTIME`，`timer_settime()` 应表现为到下一次到期的时间设置为等于 `value` 的 `it_value` 成员指定的绝对时间与 `timerid` 关联的时钟当前值之间的差值。即，当时钟达到 `value` 的 `it_value` 成员指定的值时，定时器应到期。如果指定的时间已经过去，函数应成功执行并发出到期通知。

定时器的重载值应设置为 `value` 的 `it_interval` 成员指定的值。当定时器以非零的 `it_interval` 武装时，指定了一个周期性（或重复性）定时器。

介于指定定时器分辨率的两个连续非负整数倍之间的时间值应向上舍入到较大的分辨率倍数。量化误差不应导致定时器在舍入时间值之前到期。

如果参数 `ovalue` 不为 `NULL`，`timer_settime()` 函数应在 `ovalue` 引用的位置存储一个值，表示定时器原本到期的先前时间量，如果定时器被解除武装则为零，同时包含先前的定时器重载值。定时器不应在其预定时间之前到期。

在任何时间点，对于给定的定时器，只有一个信号会被排队到进程。当一个信号仍处于挂起状态的定时器到期时，不会排队信号，并且会发生定时器超限运行（timer overrun）。当定时器到期信号被递送到或被进程接受时，`timer_getoverrun()` 函数应返回指定定时器的定时器到期超限运行计数。返回的超限运行计数包含在信号生成（排队）时间和递送或接受时间之间发生的额外定时器到期次数，最多但不包括实现定义的最大值 {DELAYTIMER_MAX}。如果此类额外到期次数大于或等于 {DELAYTIMER_MAX}，则超限运行计数应设置为 {DELAYTIMER_MAX}。`timer_getoverrun()` 返回的值适用于定时器的最近一次到期信号递送或接受。如果没有为定时器递送到期信号，`timer_getoverrun()` 的返回值是未指定的。

如果传递给 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID，则行为是未定义的。

RETURN VALUE 返回值

如果 `timer_getoverrun()` 函数成功，它应返回定时器到期超限运行计数，如上所述。

如果 `timer_gettime()` 或 `timer_settime()` 函数成功，应返回值 0。

如果这些函数中的任何一个发生错误，应返回值 -1，并设置 `errno` 以指示错误。

ERRORS 错误

`timer_settime()` 函数应在以下情况下失败：

- **[EINVAL]** `value` 结构指定的纳秒值小于零或大于或等于 1000 百万，且该结构的 `it_value` 成员未指定零秒和零纳秒。

`timer_settime()` 函数可能在以下情况下失败：

- [EINVAL] `value` 的 `it_interval` 成员不为零，且定时器是通过创建新线程（`sigev_sigev_notify` 为 SIGEV_THREAD）来创建通知的，并且 `sigev_notify_attributes` 指向的线程属性中设置了固定栈地址。

APPLICATION USAGE 应用程序使用

当定时器到期通过创建新线程来发出信号时，使用固定栈地址是有问题的。由于不能假设为一个到期创建的线程在定时器的下一次到期之前已经完成，可能会发生两个线程同时使用相同内存作为栈的情况。这是无效的，并产生未定义的结果。

RATIONALE 基本原理

实际的时钟以有限的速率滴答，100 赫兹和 1000 赫兹的速率很常见。此滴答速率的倒数是时钟分辨率，也称为时钟粒度，在任一情况下都表示为时间持续时间，对于这些常见速率分别为 10 毫秒和 1 毫秒。实际时钟的粒度意味着如果快速连续两次读取给定时钟，可能两次获得相同的时间值；并且定时器必须等待理论到期时间之后的下一个时钟滴答，以确保定时器永远不会过早返回。还要注意的是，时钟的粒度可能比用于设置和获取时间及间隔值的数据格式的分辨率明显粗糙。还要注意的是，一些实现可能选择调整时间和/或间隔值以精确匹配底层时钟的滴答。

POSIX.1-2024 卷定义了允许应用程序确定实现支持的时钟分辨率的函数，并要求实现记录定时器和 `nanosleep()` 支持的分辨率（如果它们与支持的时钟分辨率不同）。这更多是一个采购问题，而不是运行时应用程序问题。

如果实现检测到传递给 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID，建议该函数应该失败并报告 [EINVAL] 错误。

SEE ALSO 参见

- `clock_getres()`
- `timer_create()`
- `<time.h>`

CHANGE HISTORY 更改历史

首次发布于 Issue 5。为与 POSIX 实时扩展对齐而包含在内。

Issue 6

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数被标记为定时器选项的一部分。

[ENOSYS] 错误条件已被删除，因为如果实现不支持定时器选项，不需要提供存根。

[EINVAL] 错误条件已更新，包含以下内容："并且该结构的 `it_value` 成员未指定零秒和零纳秒。"此更改是针对 IEEE PASC Interpretation 1003.1 #89。

`timer_getoverrun()` 的描述已更新，以阐明："如果定时器没有递送到期信号，或者不支持实时信号扩展，`timer_getoverrun()` 的返回值是未指定的"。

`restrict` 关键字被添加到 `timer_settime()` 原型中，以与 ISO/IEC 9899:1999 标准对齐。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/140，更新 ERRORS 部分，使强制的 [EINVAL] 错误 ("`timerid` 参数不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的 ID") 变为可选。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/141，更新 ERRORS 部分，为通知方法设置为 SIGEV_THREAD 的定时器情况包含可选的 [EINVAL] 错误。还添加了应用程序使用文本。

Issue 7

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数从定时器选项移至基础部分。

与实时信号扩展选项相关的功能移至基础部分。

应用 POSIX.1-2008, Technical Corrigendum 2, XSH/TC2-2008/0370 [659]。

1.261. timer_getoverrun, timer_gettime, timer_settime — 每进程定时器

概要

```
#include <time.h>

int timer_getoverrun(timer_t timerid);
int timer_gettime(timer_t timerid, struct itimerspec *value);
int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *restrict value,
                  struct itimerspec *restrict ovalue);
```

描述

`timer_gettime()` 函数应将指定定时器 `timerid` 到期前的剩余时间量以及定时器的重载值存储到 `value` 参数指向的空间中。该结构的 `it_value` 成员应包含定时器到期前的时间量，如果定时器已解除武装则为零。即使定时器是以绝对时间武装的，该值也作为定时器到期前的间隔返回。`value` 的 `it_interval` 成员应包含由 `timer_settime()` 最后设置的重载值。

`timer_settime()` 函数应从 `value` 参数的 `it_value` 成员设置由 `timerid` 指定的定时器下次到期的时间，如果 `value` 的 `it_value` 成员非零，则武装该定时器。如果在调用 `timer_settime()` 时指定的定时器已经被武装，此调用应将下次到期的时间重置为指定的 `value`。如果 `value` 的 `it_value` 成员为零，定时器应被解除武装。对于有待处理到期通知的定时器进行解除武装或重置的效果是未指定的。

如果在参数 `flags` 中未设置 `TIMER_ABSTIME` 标志，`timer_settime()` 的行为应如同下次到期时间被设置为等于 `value` 的 `it_value` 成员指定的间隔。也就是说，定时器将在调用发起后的 `it_value` 纳秒时到期。如果在参数 `flags` 中设置了 `TIMER_ABSTIME` 标志，`timer_settime()` 的行为应如同下次到期时间被设置为等于 `value` 的 `it_value` 成员指定的绝对时间与与 `timerid` 关联的时钟当前值之间的差值。也就是说，当时钟达到 `value` 的 `it_value` 成员指定的值时，定时器将到期。如果指定的时间已经过去，函数应成功执行并发出到期通知。

定时器的重载值应设置为 `value` 的 `it_interval` 成员指定的值。当定时器以非零的 `it_interval` 武装时，指定了一个周期性（或重复性）定时器。

在指定定时器分辨率的两个连续非负整数倍之间的时间值应向上舍入到较大的分辨率倍数。量化误差不应导致定时器在舍入时间值之前到期。

如果参数 `ovalue` 不为 `NULL`，`timer_settime()` 函数应在 `ovalue` 引用的位置存储一个值，表示定时器原本到期前的时间量，如果定时器已解除武装则为零，以及之前的定时器重载值。定时器不应用于其预定时间之前到期。

在任何时间点，对于给定的定时器，只有一个信号会被排队到进程。当信号仍然处于待处理状态的定时器到期时，不会排队信号，并且会发生定时器超限运行。当定时器到期信号被递送或被进程接受时，`timer_getoverrun()` 函数应返回指定定时器的定时器到期超限计数。返回的超限计数包含在信号生成（排队）之时与信号被递送或接受之时之间发生的额外定时器到期次数，最大可达但不包括实现定义的最大值 `{DELAYTIMER_MAX}`。如果此类额外到期次数大于或等于 `{DELAYTIMER_MAX}`，则超限计数应设置为 `{DELAYTIMER_MAX}`。

`timer_getoverrun()` 返回的值适用于定时器的最近一次到期信号递送或接受。如果定时器没有递送过到期信号，`timer_getoverrun()` 的返回值是未指定的。

如果 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID，则行为是未定义的。

返回值

如果 `timer_getoverrun()` 函数成功，它应返回如上所述的定时器到期超限计数。

如果 `timer_gettime()` 或 `timer_settime()` 函数成功，应返回值 0。

如果这些函数中的任何一个发生错误，应返回值 -1，并设置 `errno` 以指示错误。

错误

`timer_settime()` 函数在以下情况下应失败：

- `EINVAL`
- 指定的 `value` 结构指定了一个小于零或大于等于 1000 百万的纳秒值，且该结构的 `it_value` 成员未指定零秒和零纳秒。

`timer_settime()` 函数在以下情况下可能失败：

- `EINVAL`

- `value` 的 `it_interval` 成员不为零，且定时器是通过创建新线程（`sigev_sigev_notify` 为 `SIGEV_THREAD`）的方式创建的通知，并且在 `sigev_notify_attributes` 指向的线程属性中设置了固定的堆栈地址。

应用程序使用

当定时器到期通过创建新线程来发信号时，使用固定堆栈地址是有问题的。由于不能假设为一个到期创建的线程在定时器的下一次到期之前已经完成，因此可能发生两个线程同时使用同一内存作为堆栈的情况。这是无效的，并会产生未定义的结果。

基本原理

实际的时钟以有限速率跳动，100 赫兹和 1000 赫兹的速率是常见的。此跳动速率的倒数就是时钟分辨率，也称为时钟粒度，在这两种情况下都表示为时间持续时间，对于这些常见速率分别为 10 毫秒和 1 毫秒。实际时钟的粒度意味着如果快速连续两次读取给定时钟，可能会两次得到相同的时间值；并且定时器必须在理论到期时间之后等待下一个时钟跳动，以确保定时器永远不会过早返回。还应注意，时钟的粒度可能比用于设置和获取时间及间隔值的数据格式的分辨率明显粗糙。另请注意，一些实现可能会选择调整时间和/或间隔值以精确匹配底层时钟的跳动。

POSIX.1-2024 卷定义了允许应用程序确定实现支持的时钟分辨率的函数，并要求实现记录定时器和 `nanosleep()` 支持的分辨率（如果它们与支持的时钟分辨率不同）。这更多是一个采购问题而不是运行时应用程序问题。

如果实现检测到 `timer_getoverrun()`、`timer_gettime()` 或 `timer_settime()` 的 `timerid` 参数指定的值不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的定时器 ID，建议函数应失败并报告 [EINVAL] 错误。

另见

- `clock_getres()`
- `timer_create()`
- `<time.h>`

变更历史

首次发布于 Issue 5。为与 POSIX 实时扩展对齐而包含在内。

Issue 6

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数被标记为定时器选项的一部分。

[ENOSYS] 错误条件已被删除，因为如果实现不支持定时器选项，不需要提供存根。

[EINVAL] 错误条件更新为包含以下内容："且该结构的 `it_value` 成员未指定零秒和零纳秒。"此更改是为了 IEEE PASC 解释 1003.1 #89。

`timer_getoverrun()` 的描述已更新，以阐明"如果定时器没有递送过到期信号，或者不支持实时信号扩展，`timer_getoverrun()` 的返回值是未指定的"。

为与 ISO/IEC 9899:1999 标准对齐，将 `restrict` 关键字添加到 `timer_settime()` 原型中。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/140，更新 ERRORS 部分，使强制的 [EINVAL] 错误 ("`timerid` 参数不对应于由 `timer_create()` 返回但尚未被 `timer_delete()` 删除的 ID") 变为可选的。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/141，更新 ERRORS 部分，为通知方法设置为 SIGEV_THREAD 时创建的定时器的情况包含可选的 [EINVAL] 错误。还添加了应用程序使用文本。

Issue 7

`timer_getoverrun()`、`timer_gettime()` 和 `timer_settime()` 函数从定时器选项移至基础。

与实时信号扩展选项相关功能被移至基础。

应用 POSIX.1-2008，技术更正 2，XSH/TC2-2008/0370 [659]。

1.262. `tolower`, `tolower_l` – 将大写字符转换为小写字符

SYNOPSIS (概要)

```
#include <ctype.h>

int tolower(int c);

[CX] int tolower_l(int c, locale_t locale);
```

DESCRIPTION (描述)

对于 `tolower()`：[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 标准遵循 ISO C 标准。

`tolower()` [CX] 和 `tolower_l()` 函数的域为 `int` 类型，其值可以表示为 `unsigned char` 或 EOF 的值。如果参数具有任何其他值，则行为未定义。如果 `tolower()` [CX] 或 `tolower_l()` 的参数表示一个大写字母，并且在当前区域 [CX] 或由 `locale` 表示的区域（类别 `LC_CTYPE`）中分别存在相应的小写字母（如字符类型信息所定义），则结果应为相应的小写字母。域中所有其他参数均原样返回。

[CX] 如果 `tolower_l()` 的 `locale` 参数是特殊的区域对象 `LC_GLOBAL_LOCALE` 或不是有效的区域对象句柄，则行为未定义。

RETURN VALUE (返回值)

成功完成后，`tolower()` [CX] 和 `tolower_l()` 函数应返回与传入参数相对应的小写字母；否则，它们应原样返回该参数。

ERRORS (错误)

未定义错误。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

无。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- `setlocale()`
- `uselocale()`
- XBD 7. Locale (区域设置)
- `<ctype.h>`
- `<locale.h>`

CHANGE HISTORY (变更历史)

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 7

从 The Open Group Technical Standard, 2006, Extended API Set Part 4 中添加了 `tolower_l()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0671 [283] 和 XSH/TC1-2008/0672 [283]。

1.263. toupper, toupper_l — 将小写字符转换为大写字符

概要

```
#include <ctype.h>

int toupper(int c);

/* XSI 选项 */
int toupper_l(int c, locale_t locale);
```

描述

对于 `toupper()` :

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 遵从 ISO C 标准。

`toupper()` 和 `toupper_l()` 函数的定义域为 `int` 类型, 其值可以表示为 `unsigned char` 或 EOF 的值。如果参数具有任何其他值, 则行为是未定义的。

如果 `toupper()` 或 `toupper_l()` 的参数表示一个小写字母, 并且在当前语言环境中或由 `locale` 表示的语言环境中 (类别 `LC_CTYPE`) 分别存在对应的大写字母, 则结果应为对应的大写字母。

定义域中的所有其他参数均保持不变返回。

如果 `toupper_l()` 的 `locale` 参数是特殊的语言环境对象 `LC_GLOBAL_LOCALE` 或不是有效的语言环境对象句柄, 则行为是未定义的。

返回值

成功完成后, `toupper()` 和 `toupper_l()` 应返回与传入参数对应的大写字母; 否则, 应返回未更改的参数。

错误

未定义错误。

示例

无。

应用程序用法

无。

基本原理

无。

未来方向

无。

另请参阅

- [setlocale\(\)](#)
- [uselocale\(\)](#)
- XBD 7. 语言环境
- [<ctype.h>](#)
- [<locale.h>](#)

变更历史

首次发布于 Issue 1。派生自 SVID 的 Issue 1。

Issue 6

标记了超出 ISO C 标准的扩展。

Issue 7

应用了 SD5-XSH-ERN-181，阐明了返回值部分。

从 The Open Group 技术标准 2006 年扩展 API 集第 4 部分添加了 `toupper_l()` 函数。

应用了 POSIX.1-2008, 技术勘误表 1, XSH/TC1-2008/0673 [283] 和 XSH/TC1-2008/0674 [283]。

1.264. tzset - 设置时区转换信息

概要

```
#include <time.h>

[XSI] extern int daylight;
[XSI] extern long timezone;

[CX] extern char *tzname[2];
[CX] void tzset(void);
```

描述

`tzset()` 函数应使用环境变量 `TZ` 的值来设置 `ctime()`、`localtime()`、`mktime()` 和 `strftime()` 函数使用的时间转换信息。如果环境中没有 `TZ`，则应使用实现定义的默认时区信息。

`tzset()` 函数应按如下方式设置外部变量 `tzname`：

```
tzname[0] = "std";
tzname[1] = "dst";
```

其中 `std` 和 `dst` 的描述如 XBD 第 8 章《环境变量》中所述。

[XSI] 如果所使用的时区永远不应用夏令时转换，`tzset()` 函数还应将外部变量 `daylight` 设置为 0；否则设置为非零值。外部变量 `timezone` 应设置为协调世界时 (UTC) 与本地标准时间之间的差值，以秒为单位。

如果一个线程在另一个线程正在调用 `tzset()` 或任何被要求或允许像调用 `tzset()` 一样设置时区信息的函数时，直接访问 `tzname`、[XSI] `daylight` 或 `timezone`，则其行为未定义。

返回值

`tzset()` 函数不应返回任何值。

错误

未定义任何错误。

示例

下表给出了示例 `TZ` 变量及其时区差值：

<code>TZ</code>	<code>timezone</code>
EST5EDT	5*60*60
GMT0	0*60*60
JST-9	-9*60*60
MET-1MEST	-1*60*60
MST7MDT	7*60*60
PST8PDT	8*60*60

应用用法

由于 `ctime()`、`localtime()`、`mktime()`、`strftime()` 和 `strftime_l()` 函数被要求像调用 `tzset()` 一样设置时区信息，因此在使用这些函数之前不需要显式调用 `tzset()`。但是，可移植应用程序在使用 `localtime_r()` 之前应显式调用 `tzset()`，因为该函数设置时区信息是可选的。

原理

无。

未来方向

无。

参见

- `ctime()`
- `localtime()`
- `mktime()`
- `strftime()`
- XBD 8. 环境变量
- `<time.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 6

更正了示例。

Issue 7

应用了 POSIX.1-2008 Technical Corrigendum 2 , XSH/TC2-2008/0377 [880]。

Issue 8

应用了 Austin Group Defect 1253 , 将"Daylight Savings"更改为"Daylight Saving"。

应用了 Austin Group Defect 1410, 移除了 `ctime_r()` 函数。

1.265. uname

概要

```
#include <sys/utsname.h>

int uname(struct utsname *name);
```

描述

`uname()` 函数应将标识当前系统的信息存储在 `name` 所指向的结构中。

`uname()` 函数使用在 `<sys/utsname.h>` 中定义的 `utsname` 结构。

`uname()` 函数应在字符数组 `sysname` 中返回命名当前系统的字符串。类似地，`nodename` 应包含此节点在实现定义的通信网络中的名称。数组 `release` 和 `version` 应进一步标识操作系统。数组 `machine` 应包含标识系统正在运行的硬件的名称。

每个成员的格式是实现定义的。

返回值

成功完成后，`uname()` 应返回 0；否则，应返回 -1 并设置 `errno` 以指示错误。

错误

未定义错误。

示例

示例 1：获取系统信息

```
#include <stdio.h>
#include <sys/utsname.h>
```

```
int main(void)
{
    struct utsname name;

    if (uname(&name) == -1) {
        perror("uname");
        return 1;
    }

    printf("System name: %s\n", name.sysname);
    printf("Node name: %s\n", name.nodename);
    printf("Release: %s\n", name.release);
    printf("Version: %s\n", name.version);
    printf("Machine: %s\n", name.machine);

    return 0;
}
```

应用程序使用

结构成员的值不受约束，与操作系统中实现的 POSIX.1-2024 本版本没有任何关系。应用程序应依赖于在 `<unistd.h>` 中定义的 `_POSIX_VERSION` 和相关常量。

POSIX.1-2024 本版本未定义结构成员的大小，并允许它们具有不同的大小，尽管大多数实现将它们定义为相同的大小：八字节加上字符串终止符的一字节。`nodename` 的大小不足以用于许多网络。

基本原理

`uname()` 函数起源于 System III、System V 和相关实现，在 Version 7 或 4.3 BSD 中不存在。在那些历史实现中，它返回的值是在系统编译时设置的。

4.3 BSD 有 `gethostname()` 和 `gethostid()`，它们分别返回符号名称和数值。还有相关的 `sethostname()` 和 `sethostid()` 函数，用于设置其他两个函数返回的值。前者包含在此规范中，后者不包含。

未来方向

无。

另见

- `gethostname()`
- `gethostid()`
- `<sys/utsname.h>`
- `<unistd.h>`

变更历史

- 首次发布于 Issue 1。派生自 System V 文档。
- 版权所有 © 2018-2024, The Open Group。保留所有权利。
SPDX-License-Identifier: CC-BY-4.0

1.266. ungetc — 将字节推回输入流

概要

```
#include <stdio.h>

int ungetc(int c, FILE *stream);
```

描述

`ungetc()` 函数应将由 `c` 指定的字节（转换为 `unsigned char`）推回到由 `stream` 指向的输入流中。推回的字节应在该流上的后续读取中以与其推入相反的顺序返回。在该流上成功调用文件定位函数（`fseek()`、`fseeko()`、`fsetpos()` 或 `rewind()`）或 `fflush()`（以 `stream` 指向的流为参数）应丢弃该流的任何推回字节。与该流对应的外部存储应保持不变。

应提供一个字节的推回功能。如果在同一流上调用 `ungetc()` 过多次数而中间没有对该流进行读取或文件定位操作，则操作可能会失败。

如果 `c` 的值等于宏 `EOF` 的值，则操作应失败且输入流应保持不变。

成功调用 `ungetc()` 应清除该流的文件结束指示符。该流的文件位置指示符应通过每次成功调用 `ungetc()` 而递减；如果在调用前其值为 0，则调用后其值是未指定的。在所有推回字节都被读取后，文件位置指示符的值应与字节被推回之前的值相同。

返回值

成功完成时，`ungetc()` 应返回转换后推回的字节。否则，应返回 `EOF`。

错误

未定义任何错误。

应用用法

无。

原理

ISO C 标准包含以下文本：“在读取或丢弃所有推回字符后，流的文件位置指示符的值应与字符被推回之前的值相同。” POSIX.1 从中省略了“或丢弃”，因为它是多余的——在 ISO C 标准中，丢弃操作由文件定位函数完成，并且不影响这些函数设置的位置。特别是，使用 SEEK_CUR 的 `fseek()` 或 `fseeko()` 的相对搜索会相对于函数入口时的位置调整位置，而不是相对于推回字节被丢弃后的位置。POSIX.1 还要求在 ISO C 标准说 `fflush()` 行为未定义的情况下，`fflush()` 丢弃推回字节。

未来方向

ISO C 标准规定，在文件位置指示符在调用前为零的二进制流上使用 `ungetc()` 是一个过时功能。在 POSIX.1 中，二进制流和文本流没有区别，因此这适用于所有流。此功能可能会在本标准的未来版本中被移除。

另请参见

- 2.5 标准I/O流
- `fseek()`
- `getc()`
- `fsetpos()`
- `read()`
- `rewind()`
- `setbuf()`
- XBD `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 7

应用了 POSIX.1-2008、技术勘误 1、XSH/TC1-2008/0687 [87,93]、XSH/TC1-2008/0688 [87] 和 XSH/TC1-2008/0689 [14]。

Issue 8

应用了 Austin Group Defect 701，阐明了流的文件位置指示符如何更新。

应用了 Austin Group Defect 1302，更改了未来方向部分。

1.267. unsetenv

SYNOPSIS

```
#include <stdlib.h>

int unsetenv(const char *name);
```

DESCRIPTION

`unsetenv()` 函数应从调用进程的环境中删除一个环境变量。`name` 参数指向一个字符串，该字符串是要删除的变量的名称。命名参数不得包含 '=' 字符。如果指定的变量在当前环境中不存在，环境应保持不变，该函数应被视为已成功完成。

`unsetenv()` 函数应更新 `environ` 指向的指针列表。

`unsetenv()` 函数不需要是线程安全的。

RETURN VALUE

成功完成后，应返回零。否则，应返回 -1，设置 `errno` 以指示错误，且环境应保持不变。

ERRORS

`unsetenv()` 函数应在以下情况下失败：

- **[EINVAL]**
- `name` 参数指向空字符串，或指向包含 '=' 字符的字符串。

EXAMPLES

无。

APPLICATION USAGE

无。

RATIONALE

请参考 [setenv\(\)](#) 中的 RATIONALE 部分。

FUTURE DIRECTIONS

无。

SEE ALSO

- [getenv\(\)](#)
- [setenv\(\)](#)
- XBD [<stdlib.h>](#), [<sys/types.h>](#)

CHANGE HISTORY

首次发布于 Issue 6。源于 IEEE P1003.1a 草案标准。

Issue 7

应用了 Austin Group Interpretation 1003.1-2001 #156。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0698 [167] 和 XSH/TC1-2008/0699 [185]。

1.268. va_arg

SYNOPSIS (概要)

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

DESCRIPTION (描述)

参考 XBD [`<stdarg.h>`](#)

[返回页面顶部](#)

1.269. va_copy

SYNOPSIS

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

DESCRIPTION

参考 XBD

1.270. va_end

SYNOPSIS

```
#include <stdarg.h>

void va_end(va_list ap);
```

DESCRIPTION

`va_end()` 宏用于在遍历可变参数列表后进行清理。它使 `va_list` 对象 `ap` 失效（除非再次调用 `va_start()` 或 `va_copy()`）。

本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意为之。POSIX.1-2024 卷遵从 ISO C 标准。

REQUIREMENTS

对 `va_start()` 和 `va_copy()` 宏的每次调用都必须在同一个函数中有对应的 `va_end()` 宏调用。

USAGE RULES

- 对象 `ap` 可以作为参数传递给另一个函数
- 如果该函数使用参数 `ap` 调用 `va_arg()` 宏，则在调用函数中 `ap` 的值是未指定的，在对 `ap` 进行任何进一步引用之前，应将其传递给 `va_end()` 宏
- 在没有对同一 `ap` 进行中间 `va_end()` 宏调用的情况下，不得调用 `va_copy()` 或 `va_start()` 宏来重新初始化 `ap`

EXAMPLE

```
#include <stdarg.h>
#include <unistd.h>
#include <stdio.h>
```

```
#define MAXARGS 31

/*
 * execl 通过以下方式调用
 * execl(file, arg1, arg2, ..., (char *)0);
 */
int execl(const char *file, const char *args, ...)
{
    va_list ap;
    char *array[MAXARGS + 1];
    int argno = 0;

    va_start(ap, args);
    while (args != 0 && argno < MAXARGS)
    {
        array[argno++] = args; // 修正了原代码中的语法错误
        args = va_arg(ap, const char *);
    }
    array[argno] = (char *) 0;
    va_end(ap);
    return execv(file, array);
}
```

RELATED FUNCTIONS

- [va_start\(\)](#) - 初始化可变参数列表
- [va_arg\(\)](#) - 从可变参数列表中检索下一个参数
- [va_copy\(\)](#) - 复制可变参数列表状态

SEE ALSO

- [printf\(\)](#)
 - [execl\(\)](#)
-

1.271. va_start, va_arg, va_copy, va_end - 处理可变参数列表

概要

```
#include <stdarg.h>

type va_arg(va_list ap, type);
void va_copy(va_list dest, va_list src);
void va_end(va_list ap);
void va_start(va_list ap, argN);
```

描述

`<stdarg.h>` 头文件应包含一组宏，允许编写接受可变参数列表的可移植函数。具有可变参数列表的函数（如 `printf()`）如果不使用这些宏，则本质上是非可移植的，因为不同系统使用不同的参数传递约定。

`<stdarg.h>` 头文件应为用于遍历列表的变量定义 `va_list` 类型。

va_start()

在调用任何 `va_arg()` 之前，应调用 `va_start()` 宏将 `ap` 初始化为列表的开始位置。

参数 `argN` 是函数定义中可变参数列表中最右边参数的标识符（即 `...` 之后的那一个）。如果参数 `argN` 使用 `register` 存储类别声明、具有函数类型或数组类型，或者具有与默认参数提升后所得类型不兼容的类型，则行为是未定义的。

va_copy()

`va_copy()` 宏将 `dest` 初始化为 `src` 的副本，就像对 `dest` 应用了 `va_start()` 宏，然后使用与之前用来达到 `src` 当前状态相同序列的 `va_arg()` 宏一样。在没有对相同的 `dest` 进行中间调用 `va_end()` 宏的情况下，不应调用 `va_copy()` 或 `va_start()` 宏来重新初始化 `dest`。

va_arg()

`va_arg()` 宏应返回 `ap` 所指向列表中的下一个参数。每次调用 `va_arg()` 都会修改 `ap`，以便依次返回连续参数的值。`type` 参数应是一个类型名称，其指定方式使得通过在 `type` 后后缀 '*' 可以简单地获得指向具有指定类型对象的指针类型。

如果没有实际的下一个参数，或者 `type` 与实际下一个参数的类型（根据默认参数提升进行提升后）不兼容，则行为是未定义的，但以下情况除外：

- 一种类型是有符号整数类型，另一种类型是相应的无符号整数类型，且值在两种类型中都可表示。
- 一种类型是指向 `void` 的指针，另一种类型是指向字符类型的指针。
- [XSI] 两种类型都是指针。

可以混合使用不同的类型，但由例程来知道期望的参数类型。

va_end()

`va_end()` 宏用于清理；它使 `ap` 无效（除非再次调用 `va_start()` 或 `va_copy()`）。

用法

在同一个函数中，每次调用 `va_start()` 和 `va_copy()` 宏都应与对应的 `va_end()` 宏调用相匹配。

可以进行多次遍历，每次都由 `va_start()` ... `va_end()` 括起来。

对象 `ap` 可以作为参数传递给另一个函数；如果该函数使用参数 `ap` 调用 `va_arg()` 宏，则调用函数中 `ap` 的值是未指定的，并且在对 `ap` 的任何进一步引用之前应将其传递给 `va_end()` 宏。

示例

这是 `exec()` 的一个可能实现：

```
#include <stdarg.h>

#define MAXARGS 31

/*
```

```
* execl 的调用方式为
* execl(file, arg1, arg2, ..., (char *) (0));
*/
int execl(const char *file, const char *args, ...)
{
    va_list ap;
    char *array[MAXARGS + 1];
    int argno = 0;

    va_start(ap, args);
    while (args != 0 && argno < MAXARGS)
    {
        array[argno++] = args;
        args = va_arg(ap, const char *);
    }
    array[argno] = (char *) 0;
    va_end(ap);
    return execv(file, array);
}
```

参见

- [printf\(\)](#)
 - [execl\(\)](#)
 - [execv\(\)](#)
-

1.272. vfprintf

概要

```
#include <stdarg.h>
#include <stdio.h>

[CX] int vasprintf(char **restrict ptr, const char *restrict fo
                  va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list a
int vfprintf(FILE *restrict stream, const char *restrict format
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_

```

描述

[CX] 本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

[CX] `vasprintf()`、`vdprintf()`、`vfprintf()`、`vprintf()`、`vsnprintf()` 和 `vsprintf()` 函数应分别等同于 [CX] `asprintf()`、`dprintf()`、`fprintf()`、`printf()`、`snprintf()` 和 `sprintf()` 函数，不同之处在于它们不是使用可变数量的参数调用，而是使用由 `<stdarg.h>` 定义的参数列表调用。

这些函数不应调用 `va_end` 宏。由于这些函数调用了 `va_arg` 宏，返回后 `ap` 的值是未指定的。

返回值

参考 `fprintf()`。

错误

参考 `fprintf()`。

以下部分为补充信息。

示例

无。

应用程序使用

使用这些函数的应用程序应在之后调用 `va_end(ap)` 进行清理。

原理

无。

未来方向

无。

另请参阅

[2.5 标准 I/O 流, `fprintf\(\)`](#)

XBD `<stdarg.h>` , `<stdio.h>`

变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了 `vsnprintf()` 函数。

Issue 6

`vfprintf()`、`vprintf()`、`vsnprintf()` 和 `vsprintf()` 函数已更新以与 ISO/IEC 9899:1999 标准对齐。

Issue 7

添加了 `vdprintf()` 函数，以补充来自 The Open Group 技术标准 2006 年扩展 API 集第 1 部分的 `dprintf()` 函数。

应用了 POSIX.1-2008 技术勘误 1，XSH/TC1-2008/0703 [14]。

Issue 8

应用了 Austin Group 缺陷 1496，添加了 `vasprintf()` 函数。

补充信息文本结束。

1.273. vfscanf, vscanf, vsscanf — stdarg 参数列表的格式化输入

SYNOPSIS (函数概要)

```
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *restrict stream,
             const char *restrict format,
             va_list arg);

int vscanf(const char *restrict format,
           va_list arg);

int vsscanf(const char *restrict s,
            const char *restrict format,
            va_list arg);
```

DESCRIPTION (描述)

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本 POSIX.1-2024 标准遵从 ISO C 标准。

`vscanf()`、`vfscanf()` 和 `vsscanf()` 函数应分别等同于 `scanf()`、`fscanf()` 和 `sscanf()` 函数，不同之处在于它们不是使用可变数量的参数调用，而是使用 `<stdarg.h>` 头文件中定义的参数列表调用。这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，返回后 `ap` 的值是未指定的。

RETURN VALUE (返回值)

参考 `fscanf()`。

ERRORS (错误)

参考 `fscanf()`。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序使用)

使用这些函数的应用程序应在之后调用 `va_end(ap)` 进行清理。

RATIONALE (基本原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- 2.5 标准I/O流
- `fscanf()`
- XBD `<stdarg.h>`, `<stdio.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14] 已被应用。

1.274. vprintf

SYNOPSIS (概要)

```
#include <stdarg.h>
#include <stdio.h>

int vasprintf(char **restrict ptr, const char *restrict format,
              va_list ap);

int vdprintf(int fildes, const char *restrict format, va_list a
int vfprintf(FILE *restrict stream, const char *restrict format
              va_list ap);

int vprintf(const char *restrict format, va_list ap);

int vsnprintf(char *restrict s, size_t n, const char *restrict
              va_list ap);

int vsprintf(char *restrict s, const char *restrict format, va_

```

◀ ▶

DESCRIPTION (描述)

本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`vasprintf()`、`vdprintf()`、`vfprintf()`、`vprintf()`、
`vsnprintf()` 和 `vsprintf()` 函数应分别等价于 `asprintf()`、
`dprintf()`、`fprintf()`、`printf()`、`snprintf()` 和 `sprintf()` 函数，不同之处在于它们不是使用可变数量的参数调用，而是使用由 `<stdarg.h>` 定义的参数列表调用。

这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，返回后 `ap` 的值是未指定的。

RETURN VALUE (返回值)

参考 `fprintf()`。

ERRORS (错误)

参考 `fprintf()`。

以下章节为补充信息。

EXAMPLES (示例)

无。

APPLICATION USAGE (应用程序用法)

使用这些函数的应用程序应在之后调用 `va_end(ap)` 进行清理。

RATIONALE (原理)

无。

FUTURE DIRECTIONS (未来方向)

无。

SEE ALSO (参见)

- 2.5 标准输入输出流
- `fprintf()`
- XBD `<stdarg.h>`, `<stdio.h>`

CHANGE HISTORY (变更历史)

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了 `vsnprintf()` 函数。

Issue 6

`vfprintf()`、`vprintf()`、`vsnprintf()` 和 `vsprintf()` 函数已更新，以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

添加了 `vdprintf()` 函数，以补充 The Open Group Technical Standard, 2006, Extended API Set Part 1 中的 `dprintf()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14]。

Issue 8

应用了 Austin Group Defect 1496，添加了 `vasprintf()` 函数。

补充信息结束。

1.275. vfscanf, vscanf, vsscanf — 格式化输入 stdarg 参数列表

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *restrict stream, const char *restrict format,
int vscanf(const char *restrict format, va_list arg);
int vsscanf(const char *restrict s, const char *restrict format
```

DESCRIPTION

`vscanf()`、`vfscanf()` 和 `vsscanf()` 函数应分别等效于 `scanf()`、`fscanf()` 和 `sscanf()` 函数，不同之处在于它们不是以可变数量的参数调用，而是以 `<stdarg.h>` 头文件中定义的参数列表调用。这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，因此返回后 `ap` 的值是未指定的。

RETURN VALUE

参考 `fscanf()`。

ERRORS

参考 `fscanf()`。

EXAMPLES

无。

APPLICATION USAGE

使用这些函数的应用程序应在之后调用 `va_end(ap)` 进行清理。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准输入/输出流
- `fscanf()`
- XBD `<stdarg.h>`
- XBD `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14] 已应用。

1.276. vsnprintf

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

/* XSI 扩展 */
int vasprintf(char **restrict ptr, const char *restrict format,
              va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list a
int vfprintf(FILE *restrict stream, const char *restrict format
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format,
              va_list ap);
```

DESCRIPTION

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是非故意的。POSIX.1-2024 本卷遵循 ISO C 标准。

`vasprintf()` 、 `vdprintf()` 、 `vfprintf()` 、 `vprintf()` 、
`vsnprintf()` 和 `vsprintf()` 函数应分别等效于 `asprintf()` 、
`dprintf()` 、 `fprintf()` 、 `printf()` 、 `snprintf()` 和 `sprintf()` 函数，不同之处在于它们不是使用可变数量的参数调用，而是使用由 `<stdarg.h>` 定义的参数列表调用。

这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，返回后 `ap` 的值是未指定的。

RETURN VALUE

参考 `fprintf()` 。

ERRORS

参考 `fprintf()`。

以下章节为信息性内容。

EXAMPLES

无。

APPLICATION USAGE

使用这些函数的应用程序应随后调用 `va_end(ap)` 进行清理。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准 I/O 流
- `fprintf()`

XBD `<stdarg.h>`, `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了 `vsnprintf()` 函数。

Issue 6

更新了 `vfprintf()`、`vprintf()`、`vsnprintf()` 和 `vsprintf()` 函数，以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

添加了 `vdprintf()` 函数，以补充 The Open Group Technical Standard, 2006, Extended API Set Part 1 中的 `dprintf()` 函数。

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0703 [14]。

Issue 8

应用了 Austin Group Defect 1496，添加了 `vasprintf()` 函数。

1.277. vsprintf - 格式化 stdarg 参数列表的输出

概要

```
#include <stdarg.h>
#include <stdio.h>

/* XSI 扩展 */
int vasprintf(char **restrict ptr, const char *restrict format,
              va_list ap);
int vdprintf(int fildes, const char *restrict format, va_list a
int vfprintf(FILE *restrict stream, const char *restrict format
              va_list ap);
int vprintf(const char *restrict format, va_list ap);
int vsnprintf(char *restrict s, size_t n, const char *restrict
              va_list ap);
int vsprintf(char *restrict s, const char *restrict format, va_

```

描述

本参考页描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。POSIX.1-2024 标准卷遵循 ISO C 标准。

`vasprintf()` 、 `vdprintf()` 、 `vfprintf()` 、 `vprintf()` 、 `vsnprintf()` 和 `vsprintf()` 函数应分别等同于 `asprintf()` 、 `dprintf()` 、 `fprintf()` 、 `printf()` 、 `snprintf()` 和 `sprintf()` 函数，不同之处在于它们不是以可变数量的参数调用，而是以 `<stdarg.h>` 定义的参数列表调用。

这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，返回后 `ap` 的值是未指定的。

返回值

参考 `fprintf()`。

错误

参考 `fprintf()`。

1.277.1. 示例

无。

1.277.2. 应用程序用法

使用这些函数的应用程序之后应调用 `va_end(ap)` 进行清理。

1.277.3. 原理

无。

1.277.4. 未来方向

无。

1.277.5. 参见

- 2.5 标准 I/O 流
- `fprintf()`
- `<stdarg.h>`
- `<stdio.h>`

1.277.6. 变更历史

首次发布于 Issue 1。源自 SVID 的 Issue 1。

Issue 5

添加了 `vsnprintf()` 函数。

Issue 6

更新了 `vfprintf()`、`vprintf()`、`vsnprintf()` 和 `vsprintf()` 函数以与 ISO/IEC 9899:1999 标准保持一致。

Issue 7

添加了 `vdprintf()` 函数，以补充来自 The Open Group 技术标准 2006 年扩展 API 集第 1 部分的 `dprintf()` 函数。

应用了 POSIX.1-2008，技术勘误表 1，XSH/TC1-2008/0703 [14]。

Issue 8

应用了 Austin Group 缺陷 1496，添加了 `vasprintf()` 函数。

1.278. vscanf

SYNOPSIS

```
#include <stdarg.h>
#include <stdio.h>

int vfscanf(FILE *restrict stream,
             const char *restrict format,
             va_list arg);

int vscanf(const char *restrict format,
           va_list arg);

int vsscanf(const char *restrict s,
            const char *restrict format,
            va_list arg);
```

DESCRIPTION

[CX] 本参考页面描述的功能与 ISO C 标准保持一致。此处描述的要求与 ISO C 标准之间的任何冲突都是无意的。本卷 POSIX.1-2024 遵从 ISO C 标准。

`vscanf()`、`vfscanf()` 和 `vsscanf()` 函数应分别等价于 `scanf()`、`fscanf()` 和 `sscanf()` 函数，不同之处在于它们不是以可变数量的参数调用，而是以 `<stdarg.h>` 头文件中定义的参数列表调用。这些函数不应调用 `va_end` 宏。由于这些函数调用 `va_arg` 宏，因此返回后 `ap` 的值是未指定的。

RETURN VALUE

参考 `fscanf()`。

ERRORS

参考 `fscanf()`。

以下章节为参考信息。

EXAMPLES

无。

APPLICATION USAGE

使用这些函数的应用程序应调用 `va_end(ap)` 进行清理。

RATIONALE

无。

FUTURE DIRECTIONS

无。

SEE ALSO

- 2.5 标准 I/O 流
- `fscanf()`
- XBD `<stdarg.h>`, `<stdio.h>`

CHANGE HISTORY

首次发布于 Issue 6。源自 ISO/IEC 9899:1999 标准。

Issue 7

应用了 POSIX.1-2008, Technical Corrigendum 1, XSH/TC1-2008/0704 [14]。

1.279. write

概要

```
#include <unistd.h>

ssize_t pwrite(int fildes, const void *buf, size_t nbytes, off_t
ssize_t write(int fildes, const void *buf, size_t nbytes);
```

描述

`write()` 函数应尝试从 `buf` 指向的缓冲区中写入 `nbyte` 个字节到与打开的文件描述符 `fildes` 关联的文件中。

在执行以下任何操作之前，如果 `nbyte` 为零且文件是常规文件，`write()` 函数可以检测并返回如下所述的错误。在没有错误的情况下，或者如果未执行错误检测，`write()` 函数应返回零且没有其他结果。如果 `nbyte` 为零且文件不是常规文件，则结果未指定。

在常规文件或其他支持定位的文件上，数据的实际写入应从与 `fildes` 关联的文件偏移量所指示的文件位置开始。在 `write()` 成功返回之前，文件偏移量应增加实际写入的字节数。对于常规文件，如果最后写入字节的位置大于或等于文件长度，则文件长度应设置为此位置加一。

在不支持定位的文件上，写入应始终从当前位置开始。与此类设备关联的文件偏移量的值未定义。

如果设置了文件状态标志的 `O_APPEND` 标志，文件偏移量应在每次写入之前设置到文件末尾，并且在更改文件偏移量和写入操作之间不应发生中间的文件修改操作。

如果 `write()` 请求写入的字节数超过可用空间（例如，进程的文件大小限制或介质的物理末尾），则只写入有空间的字节数。例如，假设在达到限制之前文件中还有 20 个字节的空间。写入 512 个字节将返回 20。下一次写入非零字节数将返回失败（除非如下所述）。

如果请求会导致文件大小超过进程的软文件大小限制，并且没有任何字节的空间可写，则请求应失败 [XSI]，实现应为线程生成 `SIGXFSZ` 信号。

如果在写入任何数据之前 `write()` 被信号中断，它应返回 -1 并将 `errno` 设置为 [EINTR]。

如果在成功写入一些数据后 `write()` 被信号中断，它应返回已写入的字节数。

如果 `nbyte` 的值大于 {SSIZE_MAX}，则结果由实现定义。

在向常规文件的 `write()` 成功返回之后：

- 从文件中被该写入修改的每个字节位置的任何成功 `read()` 都应返回该位置的 `write()` 指定的数据，直到这些字节位置再次被修改为止。
- 对文件中相同字节位置的任何后续成功 `write()` 都应覆盖该文件数据。

对管道或 FIFO 的写入请求应以与常规文件相同的方式处理，但有以下例外：

- 没有与管道或 FIFO 关联的文件偏移量，因此每个写入请求都应追加到管道或 FIFO 的末尾。
- {PIPE_BUF} 字节或更少的写入请求不应与在同一管道或 FIFO 上执行写操作的其他线程的数据交错。大于 {PIPE_BUF} 字节的写入可能会与其他线程的写操作在任意边界上交错数据，无论是否设置了文件状态标志的 `O_NONBLOCK` 标志。
- 如果 `O_NONBLOCK` 标志清除，写入请求可能导致线程阻塞，但在正常完成时应返回 `nbyte`。
- 如果设置了 `O_NONBLOCK` 标志，`write()` 请求应以以下不同方式处理：
 - `write()` 函数不应阻塞线程。
 - {PIPE_BUF} 或更少字节的写入请求应具有以下效果：如果管道或 FIFO 中有足够的可用空间，`write()` 应传输所有数据并返回请求的字节数。否则，`write()` 应不传输数据并返回 -1，将 `errno` 设置为 [EAGAIN]。
 - 超过 {PIPE_BUF} 字节的写入请求应导致以下之一：
 - 当至少可以写入一个字节时，传输它能传输的内容并返回写入的字节数。当之前写入管道或 FIFO 的所有数据都被读取时，它应传输至少 {PIPE_BUF} 字节。
 - 当无法写入任何数据时，不传输数据，并返回 -1，将 `errno` 设置为 [EAGAIN]。

当尝试写入支持非阻塞写入且无法立即接受数据的文件描述符（管道或 FIFO 除外）时：

- 如果 `O_NONBLOCK` 标志清除，`write()` 应阻塞调用线程，直到可以接受数据为止。

- 如果设置了 O_NONBLOCK 标志, `write()` 不应阻塞线程。如果不阻塞线程就可以写入一些数据, `write()` 应写入它能写入的内容并返回写入的字节数。否则, 它应返回 -1 并将 `errno` 设置为 [EAGAIN]。

成功完成时, 其中 `nbyte` 大于 0, `write()` 应标记文件的最后数据修改时间和最后文件状态更改时间以供更新, 如果文件是常规文件, 文件模式的 S_ISUID 和 S_ISGID 位可能被清除。

对于常规文件, 不会发生超过在与 `fildes` 关联的打开文件描述中建立的偏移量最大值的数据传输。

如果 `fildes` 引用套接字, `write()` 应等效于没有设置任何标志的 `send()`。

[SIO] 如果设置了 O_DSYNC 位, 文件描述符上的写入 I/O 操作应按照同步 I/O 数据完整性完成来定义的方式完成。

如果设置了 O_SYNC 位, 文件描述符上的写入 I/O 操作应按照同步 I/O 文件完整性完成来定义的方式完成。

[SHM] 如果 `fildes` 引用共享内存对象, `write()` 函数的结果未指定。

[TYM] 如果 `fildes` 引用类型化内存对象, `write()` 函数的结果未指定。

`pwrite()` 函数应等效于 `write()`, 只是它在给定位置写入且不更改文件偏移量 (无论是否设置了 O_APPEND)。`pwrite()` 的前三个参数与 `write()` 相同, 外加第四个参数 `offset` 表示文件内的所需位置。尝试在不支持定位的文件上执行 `pwrite()` 将导致错误。

返回值

成功完成时, 这些函数应返回实际写入到与 `fildes` 关联的文件中的字节数。此数字不应大于 `nbyte`。否则, 应返回 -1 并设置 `errno` 以指示错误。

错误

这些函数在以下情况下应失败:

[EAGAIN]

文件既不是管道, 也不是 FIFO, 也不是套接字, 文件描述符设置了 O_NONBLOCK 标志, 且线程在 `write()` 操作中会被延迟。

[EBADF]

`fildes` 参数不是用于写入的有效文件描述符。

[EFBIG]

尝试写入超过实现定义的最大文件大小的文件，并且没有任何字节的空间可写。

[EFBIG]

尝试写入超过进程文件大小限制的文件，并且没有任何字节的空间可写。[XSI]
还应为线程生成 SIGXFSZ 信号。

[EFBIG]

文件是常规文件，`nbyte` 大于 0，且起始位置大于或等于在与 `fildes` 关联的打开文件描述中建立的偏移量最大值。

[EINTR]

写操作由于接收到信号而终止，且没有数据传输。

[EIO]

进程是后台进程组的成员，尝试写入其控制终端，TOSTOP 已设置，调用线程没有阻塞 SIGTTOU，进程没有忽略 SIGTTOU，且进程的进程组是孤立的。此错误也可能在实现定义的条件下返回。

[ENOSPC]

包含文件的设备上没有剩余的可用空间。

`pwrite()` 函数在以下情况下应失败：

[EINVAL]

文件是常规文件或块特殊文件，且 `offset` 参数为负。文件偏移量应保持不变。

[ESPIPE]

文件不支持定位。

`write()` 函数在以下情况下应失败：

[EAGAIN]

文件是管道或 FIFO，文件描述符设置了 O_NONBLOCK 标志，且线程在写操作中会被延迟。

[EAGAIN] 或 [EWOULDBLOCK]

文件是套接字，文件描述符设置了 O_NONBLOCK 标志，且线程在写操作中会被延迟。

[ECONNRESET]

尝试在未连接的套接字上进行写入。

[EPIPE]

尝试写入没有任何进程打开用于读取的管道或 FIFO，或者只有一端打开的管道或 FIFO。还应向线程发送 SIGPIPE 信号。

[EPIPE]

尝试在已关闭写入或不再连接的套接字上进行写入。在后一种情况下，如果套接字是 SOCK_STREAM 类型，还应向线程发送 SIGPIPE 信号。

这些函数在以下情况下可能失败：

[EIO]

发生了物理 I/O 错误。

[ENOBUFS]

系统中没有足够的资源来执行操作。

[ENXIO]

对不存在的设备发出请求，或者请求超出了设备的能力。

`write()` 函数在以下情况下可能失败：

[EACCES]

尝试在套接字上进行写入，但调用进程没有适当的权限。

[ENETDOWN]

尝试在套接字上进行写入，但用于到达目的地的本地网络接口已关闭。

[ENETUNREACH]

尝试在套接字上进行写入，但没有到网络的路由。

以下部分为信息性内容。

示例

从缓冲区写入

以下示例将 `buf` 指向的缓冲区中的数据写入到与文件描述符 `fd` 关联的文件中。

```
#include <sys/types.h>
#include <string.h>
...
char buf[20];
size_t nbytes;
ssize_t bytes_written;
int fd;
...
strcpy(buf, "This is a test\n");
nbytes = strlen(buf);
```

```
bytes_written = write(fd, buf, nbytes);  
...
```

应用程序用法

无。

基本原理

另请参阅 [read\(\)](#) 中的基本原理部分。

尝试写入管道或 FIFO 有几个主要特征：

- **原子性/非原子性：**如果在一个操作中写入的全部量不与任何其他线程的数据交错，则写入是原子的。当有多个写入者向单个读取者发送数据时，这很有用。应用程序需要知道预期可以原子执行的写请求的最大大小。这个最大值称为 {PIPE_BUF}。POSIX.1-2024 的这个版本没有说明大于 {PIPE_BUF} 字节的写请求是否是原子的，但要求 {PIPE_BUF} 或更少字节的写入应是原子的。
- **阻塞/立即：**只有在 O_NONBLOCK 清除时才可能阻塞。如果有足够的空间立即写入所有请求的数据，实现应该这样做。否则，调用线程可能阻塞；即暂停，直到有足够的空间可用于写入。管道或 FIFO 的有效大小（可以在一个操作中不阻塞写入的最大量）可能根据实现而动态变化，因此无法为其指定固定值。
- **完整/部分/延迟：**写请求：

```
int fildes;  
size_t nbyte;  
ssize_t ret;  
char *buf;  
  
ret = write(fildes, buf, nbyte);
```

可能返回：

完整
`ret` = `nbyte`

部分

```
`ret` < `nbyte`
```

如果 `nbyte` $\leq \{PIPE_BUF\}$ ，这永远不会发生。如果确实发生 (`nbyte` $> \{PIPE_BUF\}$)

延迟

```
`ret` = -1, `errno` = [EAGAIN]
```

此错误表示稍后的请求可能成功。它不表示一定会成功，即使 `nbyte` $\leq \{PIPE_BUF\}$

部分写入和延迟写入只有在设置 `O_NONBLOCK` 时才可能。

这些属性的关系显示在下表中：

向管道或 FIFO 写入，`O_NONBLOCK` 清除

立即可写入：	无	部分	nbyte
<code>nbyte</code> \leq	原子阻塞 nbyte	原子阻塞 nbyte	原子立即 nbyte
<code>nbyte</code> $>$	阻塞 nbyte	阻塞 nbyte	阻塞 nbyte

如果 `O_NONBLOCK` 标志清除，如果立即可写入的量小于请求的量，写请求应阻塞。如果标志被设置（通过 `fcntl()`），写请求绝不阻塞。

向管道或 FIFO 写入，`O_NONBLOCK` 设置

立即可写入：	无	部分	nbyte
<code>nbyte</code> \leq	-1, [EAGAIN]	-1, [EAGAIN]	原子 nbyte
<code>nbyte</code> $>$	-1, [EAGAIN]	$< nbyte$ 或 -1, [EAGAIN]	$\leq nbyte$ 或 -1, [EAGAIN]

当设置 `O_NONBLOCK` 时，关于部分写入没有例外。除了写入空管道或 FIFO 之外，POSIX.1-2024 的这个版本没有确切指定何时执行部分写入，因为那将需要指定实现的内部细节。当设置 `O_NONBLOCK` 且请求量大于 `{PIPE_BUF}` 时，每个应用程序都应准备好处理部分写入，就像每个应用程序都应准备好处理其他类型文件描述符上的部分写入一样。

如果可以写入任何字节则强制至少写入一个字节的意图是为了保证如果管道或 FIFO 中有任何空间，每次写入都会取得进展。如果管道或 FIFO 为空，必须写入

{PIPE_BUF} 字节；如果不是，则必须取得一些进展。

在 POSIX.1-2024 的这个版本要求返回 -1 并将 `errno` 设置为 [EAGAIN] 的情况下，大多数历史实现返回零（设置 `O_NDELAY` 标志，这是 `O_NONBLOCK` 的历史前身，但本身不在此版本的 POSIX.1-2024 中）。POSIX.1-2024 的这个版本中的错误指示被选择为应用程序可以将这些情况与文件结束区分开来。虽然 `write()` 不能接收文件结束的指示，但 `read()` 可以，并且两个函数有相似的返回值。此外，一些现有系统（例如第八版）允许零字节写入表示读者应获得文件结束指示；对于那些系统，从 `write()` 返回零值表示成功写入了文件结束指示。

允许但不要求实现对零字节的 `write()` 请求执行错误检查。

考虑了 {PIPE_MAX} 限制的概念（指示可以在单个操作中写入管道或 FIFO 的最大字节数），但被拒绝，因为这个概念会不必要地限制应用程序写入。

另请参阅 `read()` 中关于 `O_NONBLOCK` 的讨论。

写入可以相对于其他读取和写入进行序列化。如果可以（通过任何方式）证明文件数据的 `read()` 发生在该数据的 `write()` 之后，它必须反映该 `write()`，即使调用是由不同线程进行的。类似的要求适用于对同一文件位置的多个写操作。这是保证数据从 `write()` 调用传播到后续 `read()` 调用所必需的。此要求对于网络文件系统特别重要，其中一些缓存方案违反了这些语义。

请注意，这是根据 `read()` 和 `write()` 指定的。XSI 扩展 `readv()` 和 `writev()` 也遵循这些语义。不遵循这些序列化要求的新“高性能”写入类似物也会被此措辞允许。POSIX.1-2024 的这个版本也没有说明应用程序级缓存（如 `stdio` 所做的）的任何影响。

POSIX.1-2024 的这个版本没有指定返回错误后的文件偏移量的值；情况太多了。对于编程错误，如 [EBADF]，这个概念没有意义，因为没有涉及文件。对于立即检测到的错误，如 [EAGAIN]，显然指针不应更改。然而，在中断或硬件错误之后，更新的值会非常有用，并且是许多实现的行为。

POSIX.1-2024 的这个版本没有指定从多个线程对常规文件的并发写入的行为，除了每次写入都是原子的（见 2.9.7 线程与文件操作的交互）。应用程序应使用某种形式的并发控制。

POSIX.1-2024 的这个版本有意不指定除 [ESPIPE] 之外的任何与管道、FIFO 和套接字相关的 `pwrite()` 错误。

未来方向

无。

另请参阅

`chmod()`, `creat()`, `dup()`, `fcntl()`, `getrlimit()`, `lseek()`,
`open()`, `pipe()`, `read()`, `writev()`

XBD `<limits.h>`, `<sys/uio.h>`, `<unistd.h>`

更改历史

首次在 Issue 1 中发布。源自 SVID 的 Issue 1。

Issue 5

为与 POSIX 实时扩展和 POSIX 线程扩展对齐，更新了描述。

添加了大文件峰会扩展。

添加了 `pwrite()` 函数。

Issue 6

描述指出 `write()` 函数不阻塞线程。之前这说的是"进程"而不是"线程"。

为与 XSI STREAMS 选项组的一部分，在描述和错误部分更新了对 STREAMS 的引用。

从与单一 UNIX 规范对齐得出以下对 POSIX 实现的新要求：

- 描述现在指出如果 `write()` 在成功写入一些数据后被信号中断，它返回写入的字节数。在 POSIX.1-1988 标准中，`write()` 是返回写入的字节数还是返回 -1 并将 `errno` 设置为 [EINTR] 是可选的。这是 FIPS 要求。
- 为支持大文件进行了以下更改：
 - 对于常规文件，不会发生超过在与 `fildes` 关联的打开文件描述中建立的偏移量最大值的数据传输。
 - 添加了第二个 [EFBIG] 错误条件。
- 添加了 [EIO] 错误条件。
- 为管道只有一端打开的情况添加了 [EPIPE] 错误条件。
- 添加了 [ENXIO] 可选错误条件。

在描述中添加了关于套接字的文本。

为与 IEEE P1003.1a 草案标准对齐进行了以下更改：

- 澄清了零字节读取的效果。

通过指定 `write()` 对类型化内存对象的结果未指定，更新了描述以与 IEEE Std 1003.1j-2000 对齐。

为套接字上的操作添加了以下错误条件：[EAGAIN]、[EWOULDBLOCK]、[ECONNRESET]、[ENOTCONN] 和 [EPIPE]。

使 [EIO] 错误变为可选。

为套接字添加了 [ENOBUFS] 错误。

为套接字上的操作添加了以下错误条件：[EACCES]、[ENETDOWN] 和 [ENETUNREACH]。

`writev()` 函数被拆分到单独的参考页面。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/146，将错误部分中的文本从"向调用进程生成 SIGPIPE 信号"更新为"还应向线程发送 SIGPIPE 信号"。

应用 IEEE Std 1003.1-2001/Cor 2-2004，项目 XSH/TC2/D6/147，对基本原理进行了更正。

Issue 7

`pwrite()` 函数从 XSI 选项移到基础部分。

与 XSI STREAMS 选项相关的功能被标记为过时。

应用 SD5-XSH-ERN-160，更新描述以澄清 `pwrite()` 函数的要求，并将"文件指针"的使用更改为"文件偏移量"。

应用 POSIX.1-2008，技术勘误 1，XSH/TC1-2008/0742 [219]、XSH/TC1-2008/0743 [215]、XSH/TC1-2008/0744 [79] 和 XSH/TC1-2008/0745 [215]。

应用 POSIX.1-2008，技术勘误 2，XSH/TC2-2008/0401 [676,710] 和 XSH/TC2-2008/0402 [966]。

Issue 8

应用 Austin Group 缺陷 308，澄清 [EFBIG] 错误的处理。

应用 Austin Group 缺陷 1330，删除过时接口。

应用 Austin Group 缺陷 1430，澄清与管道和 FIFO 上数据交错相关的要求适用于其他线程中的写操作，而不仅仅是其他进程，并将一些"管道"的使用更改为"管道或 FIFO"。

应用 Austin Group 缺陷 1669，从与进程文件大小限制相关的 [EFBIG] 错误部分
删除 XSI 着色。
