linux进程-应用编程

# 进程基本概念

## 1程序pk进程

|  |
| --- |
| 什么是程序  程序是完成特定任务的一系列指令集合。 |
| 什么是进程  从用户的角度来看进程是程序的一次执行过程  从操作系统的核心来看，进程是操作系统分配的内存、CPU时间片等资源的基本单位。  进程是资源分配的最小单位  每一个进程都有自己独立的地址空间与执行状态。  像UNIX这样的多任务操作系统能够让许多程序同时运行，每一个运行着的程序就构成了一个进程 |
| 进程数据结构 |
| 进程的静态描述：由三部分组成:PCB、有关程序段和该程序段对其进行操作的数据结构集。  进程控制块：用于描述进程情况及控制进程运行所需的全部信息。  代码段：是进程中能被进程调度程序在CPU上执行的程序代码段。  数据段：一个进程的数据段，可以是进程对应的程序加工处理的原始数据，也可以是程序执行后产生的中间或最终数据 |
| 进程和程序的区别  进程是动态的，程序是静态的  进程的生命周期是相对短暂的，而程序是永久的。  进程数据结构PCB  一个进程只能对应一个程序，一个程序可以对应多个进程。 |
| 总结：  中断  早期是没有进程这个概念，当出现中断技术以后才出现进程这个概念  分式操作系统  基于时间片轮转  进程是操作系统对资源的一种抽象，一个进程：代码段、数据段、堆栈段、+进程控制块（PCB）  PCB是操作系统感知进程存在的一个重要数据结构（cpu通过进程控制块来控制进程） |

## 2进程状态

### 进程状态

|  |
| --- |
| 操作系统经典三态   * 就绪 * 等待（阻塞） * 运行 |
| 总结状态条件和原因 |
| 因创建而就绪，因调度而执行；因时间片用完而重新就绪；  执行中因I/O请求而阻塞；  I/O完成而就绪  注意：阻塞以后不能直接执行，必须进入就绪状态。 |
| Linux内核中的进程状态   * 运行状态（TASK\_RUNNING） * 可中断睡眠状态（TASK\_INTERRUPTIBLE） * 不可中断睡眠状态（TASK\_UNINTERRUPTIBLE） * 暂停状态（TASK\_STOPPED） * 僵死状态（TASK\_ZOMBIE） |
|  |
| 备注：就绪态深入理解：内存中就绪和交换空间中就绪，操作系统支持虚拟内存。  虚拟内存实现需要操作系统支持：内存段式管理、业式管理、段页管理。 |
| 进程状态编程事件 |

### 进程调度及调度算法

|  |
| --- |
|  |
|  |
|  |

## 3进程编程相关术语

|  |
| --- |
| 进程控制块 |
| * 进程描述信息   + 进程标识符用于唯一的标识一个进程。 * 进程控制信息   + 进程当前状态   + 进程优先级   + 程序开始地址   + 各种计时信息   + 通信信息 * 资源信息   + 占用内存大小及管理用数据结构指针   + 交换区相关信息   + I/O设备号、缓冲、设备相关的数结构   + 文件系统相关指针 * 现场保护信息   + 寄存器   + PC   + 程序状态字PSW   + 栈指针 |
| 进程标示 |
| * 每个进程都会分配到一个独一无二的数字编号，我们称之为“进程标识”(process identifier),或者就直接叫它PID. * 是一个正整数，取值范围从2到32768 * 当一个进程被启动时，它会顺序挑选下一个未使用的编号数字做为自己的PID * 数字1一般为特殊进程init保留的 |
| 进程创建 |
| * 不同的操作系统所提供的进程创建原语的名称和格式不尽相同，但执行创建进程原语后，操作系统所做的工作却大致相同，都包括以下几点： * 给新创建的进程分配一个内部标识，在内核中建立进程结构。 * 复制父进程的环境 * 为进程分配资源， 包括进程映像所需要的所有元素（程序、数据、用户栈等）， * 复制父进程地址空间的内容到该进程地址空间中。 * 置该进程的状态为就绪，插入就绪队列。 |
| 进程撤销 |
| * 进程终止时操作系统做以下工作： * **关闭软中断***:*因为进程即将终止而不再处理任何软中断信号； * **回收资源：**释放进程分配的所有资源，如关闭所有已打开文件，释放进程相应的数据结构等； * **写记帐信息**：将进程在运行过程中所产生的记帐数据（其中包括进程运行时的各种统计信息）记录到一个全局记帐文件中； * **置该进程为僵死状态**:向父进程发送子进程死的软中断信号，将终止信息status送到指定的存储单元中； * **转进程调度**:因为此时CPU已经被释放，需要由进程调度进行CPU再分配。 |
|  |
|  |
| 进程创建其他实践  1）Win PK Linux进程观察工具  0号进程（也称为）空闲进程  内存===》交换空间，支持虚拟内存。  1号进程 第一个用户进程。。。。  2）which init /sbin/  3）查看内核进程pid最大配置  [root@localhost ~]# cat /proc/sys/kernel/pid\_max  32768  4）终止进程方法中  SIGABORT |
|  |

# 进程编程实践

## 1fork系统调用相关说明

|  |
| --- |
| **复制一个进程映象fork**   * 使用fork函数得到的子进程从父进程的继承了整个进程的地址空间，包括：进程上下文、进程堆栈、内存信息、打开的文件描述符、信号控制设置、进程优先级、进程组号、当前工作目录、根目录、资源限制、控制终端等。 * 子进程与父进程的区别在于：   + 1、父进程设置的锁，子进程不继承   + 2、各自的进程ID和父进程ID不同   + 3、子进程的未决告警被清除；   + 4、子进程的未决信号集设置为空集。   **fork系统调用**   * 包含头文件 <sys/types.h> 和 <unistd.h> * 函数功能:创建一个子进程 * 函数原型   pid\_t fork(void);   * 参数：无参数。 * 返回值:   + 如果成功创建一个子进程，对于父进程来说返回子进程ID   + 如果成功创建一个子进程，对于子进程来说返回值为0   + 如果为-1表示创建失败 |
|  |
| 注意点：   * 理解1：fork系统调用之后，父子进程将交替执行。 * 理解2：怎么样理解一次调用2次返回? * 理解3：怎么样理解，fork返回值大于零的是父进程，为什么这样设计:？ * 理解4：怎么样理解分支在fork之后，而不是父进程main函数的开始？ |
|  |
| 总结：  怎么样理解，一次调用，二次返回？  问题的本质是：两次返回，是在各自的进程空间中返回的。  子进程和父进程各有自己的内存空间 （fork：代码段、数据段、堆栈段、PCB进程控制块的copy）。 |
| 总结：  #include <sys/types.h>  #include <unistd.h>  #include <sys/stat.h>  #include <fcntl.h>  #include <stdlib.h>  #include <stdio.h>  #include <string.h>  #include <signal.h>  #include <errno.h>  #include <signal.h>  int main(void )  {  int fd;  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  printf("befor fork pid:%d \n", getpid());  int num = 10; //思考打印  fd = open("11.txt", O\_WRONLY);  if (fd == -1)  {  return 0;  }    pid = fork();  if (pid == -1)  {  printf("pid < 0 err.\n");  return -1;  }  if (pid > 0)  {  printf("parent: pid:%d \n", getpid());  write(fd, "parent", 6);  close(fd);  //sleep(1);  }  else if (pid == 0)  {  printf("child: %d, parent: %d \n", getpid(), getppid());  write(fd, "child", 5);  close(fd);  //sleep(100);  }  printf("fork after....\n");  return 0;  } |
| 强化训练  //请问打印几句hello….  int main()  {  fork();  fork();  fork();  printf("hello…..\n");  return 0;  } |
| **写时复制copy on write**   * 如果多个进程要读取它们自己的那部分资源的副本，那么复制是不必要的。 * 每个进程只要保存一个指向这个资源的指针就可以了。 * 如果一个进程要修改自己的那份资源的“副本”，那么就会复制那份资源。这就是写时复制的含义 |
| 原因分析：加快速度，linux内核是段页式管理机制（因段管理从0开始，），也可叫页式管理机制。只复制对应的页。缺页，在中断查询，再赋值。  概念要深入理解 |

## 2孤儿进程和僵尸进程

|  |
| --- |
|  |
| 孤儿进程和僵尸进程   * 如果父进程先退出，子进程还没退出那么子进程的父进程将变为init进程。（注：任何一个进程都必须有父进程） * 如果子进程先退出，父进程还没退出，那么子进程必须等到父进程捕获到了子进程的退出状态才真正结束，否则这个时候子进程就成为僵进程。   孤儿进程  如果父亲进程先结束，子进程会托孤给1号进程 |
| root 14121 14089 0 22:20 pts/18 00:00:00 su - wbm01  wbm01 14122 14121 0 22:20 pts/18 00:00:00 -bash  wbm01 14300 1 0 22:38 pts/18 00:00:00 ./dm02\_fork  wbm01 14308 14028 0 22:38 pts/17 00:00:00 ps -ef  [wbm01@localhost 01process]$ ps –ef |
| 僵尸进程 |
| 如果子进程结束，父进程还没有查询子进程的状态，那么子进程就会是僵尸状态  [wbm01@localhost 01process]$ ps -u wbm01  PID TTY TIME CMD  14028 pts/17 00:00:00 bash  14122 pts/18 00:00:00 bash  14233 pts/18 00:00:00 dm02\_fork  14234 pts/18 00:00:00 dm02\_fork <defunct>  14235 pts/17 00:00:00 ps |
| 避免僵尸进程  #include <signal.h>  signal(SIGCHLD, SIG\_IGN); |

## 3fork之后父子进程共享文件

|  |
| --- |
| 父进程的文件描述符   * 如果父进程中打开了一个文件，那么子进程不需要在打开 * 会不会像int 赋值两份。。。。。赋值的文件描述符是什么意思？ |
|  |
| void main(void)  {  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  int fd;  fd = open("test.txt", O\_WRONLY);  if (fd == -1)  {  perror("open err");  exit(0);  }    pid = fork();  if (pid == -1)  {  perror("fork err");  return -1;  }  if (pid == 0)  {  printf("this is child pid:%d ppid:%d \n", getpid(), getppid());  write(fd, "child", 5);  //sleep(20);  }  if (pid > 0)  {  printf("this is parent pid:%d ppid:%d \n", getpid(), getppid());  write(fd, "parent", 5);  //sleep(20);  }  printf("fork() after\n");  sleep(1);  close(fd);  return 0;  } |
| fd = open("test.txt", O\_WRONLY);  write(fd, "parent", 5); |

## 4fork和vfork

|  |
| --- |
| 1）在fork还没实现copy on write之前。Unix设计者很关心fork之后立刻执行exec所造成的地址空间浪费，所以引入了vfork系统调用。  2）vfork有个限制，子进程必须立刻执行\_exit或者exec函数。  即使fork实现了copy on write，效率也没有vfork高，但是我们不推荐使用vfork，因为几乎每一个vfork的实现，都或多或少存在一定的问题。 |
| 结论：  1：fork子进程拷贝父进程的数据段  Vfork子进程与父进程共享数据段；  2：fork父、子进程的执行次序不确定  Vfork：子进程先运行，父进程后运行； |
|  |
| Vfork和exec函数族在一起  execve替换进程映像（加载程序）注意execve是一个系统调用;  替换意味着：代码段、数据段、堆栈段、进程控制块PCB全部替换。 |
| int main(void)  {  int fd;  pid\_t pid;  signal(SIGCHLD, SIG\_IGN);  printf("befor fork pid:%d \n", getpid());  g\_num = 10;  pid = vfork();  if (pid == -1)  {  printf("pid < 0 err.\n");  return -1;  }    if (pid > 0)  {  printf("parent: pid:%d \n", getpid());  }  else if (pid == 0)  {  printf("child: %d, parent: %d \n", getpid(), getppid());    // int execve(const char \*filename, char \*const argv[],  // char \*const envp[]);  char \*const argv[] = {"ls", "-lt", NULL};  execve("/bin/ls", argv, NULL);    //注意：此处有/bin/ls程序结束程序，不会出现coredump现象    printf("测试下面这句话还执行吗\n");  //exit(0);  //1 vfork只有需要用exit(0) \_exit(0)  //2 测试return 0; 区别  }  printf("fork after....\n");  return 0;  } |

## 5进程终止的5种方式

|  |
| --- |
| 进程终止的5种方式   * 正常退出   + 从main函数返回   + 调用exit   + 调用\_exit * 异常退出   + 调用abort 产生SIGABOUT信号   + 由信号终止 **ctrl+c SIGINT** |
|  |
| eg：  区别1：清空缓冲区的操作  int main(void)  {  printf("hello itcast");  //return 0;  //exit(0);  fflush(stdout);  \_exit(0);  }  区别2：exit会调用终止处理程序  有关终止处理程序 |
| * atexit可以注册终止处理程序，ANSI C规定最多可以注册32个终止处理程序。 * 终止处理程序的调用与注册次序相反 * man atexit   int atexit(void (\*function)(void));  man 2 atexit |
| 总结exit与\_exit区别  1）\_exit是一个系统调用，exit是一个c库函数  2）exit会执行清楚I/O缓存  3）exit会执行调用终止处理程序 |
|  |

## 6exec函数族替换进程印象

|  |
| --- |
| exec替换进程印象   * 在进程的创建上Unix采用了一个独特的方法，它将进程创建与加载一个新进程映象分离。这样的好处是有更多的余地对两种操作进行管理。 * 当我们创建了一个进程之后，通常将子进程替换成新的进程映象，这可以用exec系列的函数来进行。当然，exec系列的函数也可以将当前进程替换掉。 |
| exec关联的函数族   * 包含头文件<unistd.h> * 功能用exec函数可以把当前进程替换为一个新进程。exec名下是由多个关联函数组成的一个完整系列，头文件<unistd.h> * 原型   /\*  #include <unistd.h>  extern char \*\*environ;  int execl(const char \*path, const char \*arg, ...);  int execlp(const char \*file, const char \*arg, ...); PATH  int execle(const char \*path, const char \*arg,  ..., char \* const envp[]);  int execv(const char \*path, char \*const argv[]);  int execvp(const char \*file, char \*const argv[]);  \*/ |
| exec exec关联的函数族参考 man 手册 |
| 以execlp、execvp、和execle讲解   * path参数表示你要启动程序的名称包括路径名 * arg参数表示启动程序所带的参数 * 返回值:成功返回0,失败返回-1 |
| * execl，execlp，execle（都带“l”）的参数个数是可变的，参数以一个空指针结束。 * execv和execvp的第二个参数是一个字符串数组，新程序在启动时会把在argv数组中给定的参数传递到main * 这些函数通常都是用execve实现的，这是一种约定俗成的做法，并不是非这样不可。 |
| * 名字最后一个字母是“p”的函数会搜索PATH环境变量去查找新程序的可执行文件。如果可执行文件不在PATH定义的路径上，就必须把包括子目录在内的绝对文件名做为一个参数传递给这些函数   总结：l代表可变参数列表，p代表在path环境变量中搜索file文件。envp代表环境变量。 |
| int main(void)  {  //演示程序被完全替换  //替换以后，pid不会发生变化  //注意 printf后的\n不能忘记，不然main函数打印不出来  printf(“main getpid: %d\n”, getpid());    //execlp(“ls”, “ls”, “-lt”, NULL);  int ret = execlp(“./testpid2”, NULL, NULL);  if (ret == -1)  {  perror(“ERR: “);  }    printf(“fork after….\n”);  return 0;  } |
| 有关环境变量  int main(int argc, char \*argv[])  {  printf(“main egin…\n”);  //1)如果envp环境参数列表不填写，则testpidandenv程序会使用默认的环境参数列表  //2)如果envp环境参数列表填写，则testpidandenv程序会使用你填写的环境参数列表  char \*ars[] = {“aa=111”, “bb==222”, NULL};  int ret = execle(“./testpidandenv”,”testpidandenv”, NULL,NULL);  //int ret = execle(“./testpidandenv”,”testpidandenv”, NULL,ars);    if (ret == -1)  {  perror("main:");  }  printf("main end...\n");  return 0;  } |

## 7父进程wait和waitpid

|  |
| --- |
| 1、wait和waitpid出现的原因 |
| SIGCHLD   * 当子进程退出的时候，内核会向父进程发送SIGCHLD信号，子进程的退出是个异步事件（子进程可以在父进程运行的任何时刻终止） * 子进程退出时，内核将子进程置为僵尸状态，这个进程称为僵尸进程，它只保留最小的一些内核数据结构，以便父进程查询子进程的退出状态。   父进程查询子进程的退出状态可以用wait/waitpid函数 |
| 2、wait和waitpid函数用法 |
| Wait   * 头文件<sys/types.h>和<sys/wait.h> * 函数功能:当我们用fork启动一个进程时，子进程就有了自己的生命，并将独立地运行。有时，我们需要知道某个子进程是否已经结束了，我们可以通过wait安排父进程在子进程结束之后。 * 函数原型   + pid\_t wait(int \*status) * 函数参数   + status:该参数可以获得你等待子进程的信息 * 返回值：   + 成功等待子进程函数返回等待子进程的ID * wait系统调用会使父进程暂停执行，直到它的一个子进程结束为止。 * 返回的是子进程的PID，它通常是结束的子进程 * 状态信息允许父进程判定子进程的退出状态，即从子进程的main函数返回的值或子进程中exit语句的退出码。 * 如果status不是一个空指针，状态信息将被写入它指向的位置 |
| Wait获取status后检测处理  宏定义 描述  WIFEXITED(status) 如果子进程正常结束，返回一个非零值  WEXITSTATUS(status) 如果WIFEXITED非零，返回子进程退出码  WIFSIGNALED(status) 子进程因为捕获信号而终止，返回非零值  WTERMSIG(status) 如果WIFSIGNALED非零，返回信号代码  WIFSTOPPED(status) 如果子进程被暂停，返回一个非零值  WSTOPSIG(status) 如果WIFSTOPPED非零，返回一个信号代码 |
| Waitpid  ❑函数功能:用来等待某个特定进程的结束  ❑函数原型:  **pid\_t waitpid(pid\_t pid, int \*status,int options)**  ❑参数:   * + status:如果不是空，会把状态信息写到它指向的位置   ❑options：允许改变waitpid的行为，最有用的一个选项是WNOHANG,它的作用是防止waitpid把调用者的执行挂起   * + 返回值：如果成功返回等待子进程的ID，失败返回-1 |
| 对于waitpid的*p I d*参数的解释与其值有关：   * *pid* == -1 等待任一子进程。于是在这一功能方面waitpid与wait等效。 * *pid* > 0 等待其进程I D与*p I d*相等的子进程。 * *pid* == 0 等待其组I D等于调用进程的组I D的任一子进程。换句话说是与调用者进程同在一个组的进程。 * *pid* < -1 等待其组I D等于*p I d*的绝对值的任一子进程。 |
| 3、wait pk waitpid |
| Wait和waitpid区别和联系   * 在一个子进程终止前， wait 使其调用者阻塞，而waitpid 有一选择项，可使调用者不阻塞。 * waitpid并不等待第一个终止的子进程—它有若干个选择项，可以控制它所等待的特定进程。 * 实际上wait函数是waitpid函数的一个特例。 |
| 僵尸进程   * 当一个子进程结束运行时，它与其父进程之间的关联还会保持到父进程也正常地结束运行或者父进程调用了wait才告终止。   进程表中代表子进程的数据项是不会立刻释放的，虽然不再活跃了，可子进程还停留在系统里，因为它的退出码还需要保存起来以备父进程中后续的wait调用使用。它将称为一个“僵进程” |
| 如何避免僵尸进程 |
| * 调用wait或者waitpid函数查询子进程退出状态，此方法父进程会被挂起。 * 如果不想让父进程挂起，可以在父进程中加入一条语句：signal(SIGCHLD,SIG\_IGN);表示父进程忽略SIGCHLD信号，该信号是子进程退出的时候向父进程发送的。 |

wait和waitpid的工程实践

|  |
| --- |
| abort() //会发出一个6号信号  man 7 signal  [wbm01@localhost 03exec]$ kill -l  1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL  5) SIGTRAP 6) SIGABRT 7) SIGBUS 8) SIGFPE  9) SIGKILL 10) SIGUSR1 11) SIGSEGV 12) SIGUSR2  13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGSTKFLT  17) SIGCHLD 18) SIGCONT 19) SIGSTOP 20) SIGTSTP  21) SIGTTIN 22) SIGTTOU 23) SIGURG 24) SIGXCPU  25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH  29) SIGIO 30) SIGPWR 31) SIGSYS 34) SIGRTMIN  35) SIGRTMIN+1 36) SIGRTMIN+2 37) SIGRTMIN+3 38) SIGRTMIN+4  39) SIGRTMIN+5 40) SIGRTMIN+6 41) SIGRTMIN+7 42) SIGRTMIN+8  43) SIGRTMIN+9 44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12  47) SIGRTMIN+13 48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14  51) SIGRTMAX-13 52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10  55) SIGRTMAX-9 56) SIGRTMAX-8 57) SIGRTMAX-7 58) SIGRTMAX-6  59) SIGRTMAX-5 60) SIGRTMAX-4 61) SIGRTMAX-3 62) SIGRTMAX-2  63) SIGRTMAX-1 64) SIGRTMAX |

利用man手册，大大减少开发难度，提高开发技能

|  |
| --- |
| Man 7 signal 查找有什么信号可以使应用程序暂停  Standard Signals  Linux supports the standard signals listed below. Several signal numbers are architecture dependent, as indicated in  the "Value" column. (Where three values are given, the first one is usually valid for alpha and sparc, the middle one  for i386, ppc and sh, and the last one for mips. A - denotes that a signal is absent on the corresponding architec-  ture.)  First the signals described in the original POSIX.1-1990 standard.  Signal Value Action Comment  -------------------------------------------------------------------------  SIGHUP 1 Term Hangup detected on controlling terminal  or death of controlling process  SIGINT 2 Term Interrupt from keyboard  SIGQUIT 3 Core Quit from keyboard  SIGILL 4 Core Illegal Instruction  SIGABRT 6 Core Abort signal from abort(3)  SIGFPE 8 Core Floating point exception  SIGKILL 9 Term Kill signal  SIGSEGV 11 Core Invalid memory reference  SIGPIPE 13 Term Broken pipe: write to pipe with no readers  SIGALRM 14 Term Timer signal from alarm(2)  SIGTERM 15 Term Termination signal  SIGUSR1 30,10,16 Term User-defined signal 1  SIGUSR2 31,12,17 Term User-defined signal 2  SIGCHLD 20,17,18 Ign Child stopped or terminated  SIGCONT 19,18,25 Cont Continue if stopped  SIGSTOP 17,19,23 Stop Stop process  SIGTSTP 18,20,24 Stop Stop typed at tty  SIGTTIN 21,21,26 Stop tty input for background process  SIGTTOU 22,22,27 Stop tty output for background process  The signals SIGKILL and SIGSTOP cannot be caught, blocked, or ignored. |
| Man 2wait  SYNOPSIS  #include <sys/types.h>  #include <sys/wait.h>  pid\_t wait(int \*status);  pid\_t waitpid(pid\_t pid, int \*status, int options);  int waitid(idtype\_t idtype, id\_t id, siginfo\_t \*infop, int options);  在手册中清楚的写明了用法 |
| EXAMPLE  The following program demonstrates the use of fork(2) and waitpid(2). The program creates a child process. If no com-  mand-line argument is supplied to the program, then the child suspends its execution using pause(2), to allow the user  to send signals to the child. Otherwise, if a command-line argument is supplied, then the child exits immediately,  using the integer supplied on the command line as the exit status. The parent process executes a loop that monitors  the child using waitpid(2), and uses the W\*() macros described above to analyse the wait status value.  The following shell session demonstrates the use of the program:  例子：  $ ./a.out &  Child PID is 32360  [1] 32359  $ kill -STOP 32360  stopped by signal 19  $ kill -CONT 32360  continued  $ kill -TERM 32360  killed by signal 15  [1]+ Done ./a.out  $  #include <sys/wait.h>  #include <stdlib.h>  #include <unistd.h>  #include <stdio.h>  int  main(int argc, char \*argv[])  {  pid\_t cpid, w;  int status;  cpid = fork();  if (cpid == -1) { perror("fork"); exit(EXIT\_FAILURE); }  if (cpid == 0) { /\* Code executed by child \*/  printf("Child PID is %ld\n", (long) getpid());  if (argc == 1)  pause(); /\* Wait for signals \*/  \_exit(atoi(argv[1]));  } else { /\* Code executed by parent \*/  do {  w = waitpid(cpid, &status, WUNTRACED | WCONTINUED);  if (w == -1) { perror("waitpid"); exit(EXIT\_FAILURE); }  if (WIFEXITED(status)) {  printf("exited, status=%d\n", WEXITSTATUS(status));  } else if (WIFSIGNALED(status)) {  printf("killed by signal %d\n", WTERMSIG(status));  } else if (WIFSTOPPED(status)) {  printf("stopped by signal %d\n", WSTOPSIG(status));  } else if (WIFCONTINUED(status)) {  printf("continued\n");  }  } while (!WIFEXITED(status) && !WIFSIGNALED(status));  exit(EXIT\_SUCCESS);  }  } |

## 8system C库函数

|  |
| --- |
| System   * 功能：system()函数调用“/bin/sh -c command”执行特定的命令，阻塞当前进程直到command命令执行完毕 * 原型：   int system(const char \*command);   * 返回值：   如果无法启动shell运行命令，system将返回127；出现不能执行system调用的其他错误时返回-1。如果system能够顺利执行，返回那个命令的退出码。   * system函数执行时，会调用fork、execve、waitpid等函数。 |
| 自动动手写system命令  int my\_system(const char \*command)  {  pid\_t pid;  int status;  if (command == NULL)  return 1;  if ((pid = fork()) < 0)  status = -1;  else if (pid == 0)  {  execl("/bin/sh", "sh", "-c", command, NULL);  exit(127);  }  else  {  while (waitpid(pid, &status, 0) < 0)  {  if (errno == EINTR)  continue;    status = -1;  break;  }  }  return status;  } |

## 9守护进程

守护进程概念

|  |
| --- |
|  |
| 什么是守护进程   * 守护进程是在后台运行不受控端控制的进程，通常情况下守护进程在系统启动时自动运行 * 守护进程的名称通常以d结尾，比如sshd、xinetd、crond等 |
| 创建守护进程步骤   * 调用fork(),创建新进程，它会是将来的守护进程 * 在父进程中调用exit，保证子进程不是进程组组长 * 调用setsid创建新的会话期 * 将当前目录改为根目录 （如果把当前目录作为守护进程的目录，当前目录不能被卸载，它作为守护进程的工作目录了。） * 将标准输入、标准输出、标准错误重定向到/dev/null |
| 守护进程api  **int daemon(int nochdir, int noclose);**   * 功能：创建一个守护进程 * 参数：   + nochdir：=0将当前目录更改至“/”   + noclose：=0将标准输入、标准输出、标准错误重定向至“/dev/null” |
|  |

守护进程工程实践

|  |
| --- |
|  |
| 1）  man setsid  DESCRIPTION  setsid() creates a new session if the calling process is not a process group leader. The calling process is the leader of the new session, the process group leader of the new process group, and has no controlling tty. The process group ID and session ID of the calling process are set to the PID of the calling process. The calling process will be the only process in this new process group and in this new session. |
| Setsid创建一个新的会话；调用者进程会是这个会话期唯一的一个进程，是唯一组的组长；调用者进程id是组id，也是会话期的id。不能用进程组组长去调用setsid函数 |
| 2）  [wbm01@localhost 04wait]$ ps –ef  UID PID PPID C STIME TTY TIME CMD  wbm01 14122 14121 0 Jun25 pts/18 00:00:00 –bash  root 14431 3005 0 Jun25 ? 00:00:00 sshd: wbm01 [priv]  wbm01 14433 14431 0 Jun25 ? 00:00:00 sshd: wbm01@notty  wbm01 14434 14433 0 Jun25 ? 00:00:00 /usr/libexec/openssh/sftp-server  root 15064 1583 0 Jun25 pts/1 00:00:00 su – wbm01  wbm01 15065 15064 0 Jun25 pts/1 00:00:00 –bash  wbm01 25328 14028 0 12:56 pts/17 00:00:00 ps –ef  有的进程有tty，有的进程没有tty。带问号的是守护进程。 |
| 3）  可以利用killall 进程名字 来杀一族进程，命令很爽。。。。不需要 kill -9 pid  root 15064 1583 0 Jun25 pts/1 00:00:00 su – wbm01  wbm01 15065 15064 0 Jun25 pts/1 00:00:00 –bash  wbm01 25635 1 70 13:20 ? 00:00:16 ./dm01\_mydaemon  wbm01 25674 1 42 13:20 ? 00:00:03 ./dm01\_mydaemon  wbm01 25676 14028 0 13:20 pts/17 00:00:00 ps –ef  [wbm01@localhost 05daemon]$ killall dm01\_mydaemon  [wbm01@localhost 05daemon]$ pwd  /home/wbm01/05daemon |
| 4）守护进程，关闭和不关闭重定向之间的区别。。。  做实验。。。。。 |
| 5、其他知识延伸  会话期、进程组 |
|  |
| 会话期：是一个或者多个进程组的集合，通常一个会话期开始与用户登录，终止于用户退出。在此期间，该用户运行的所有进程都属于这个会话期。 |
| 请你思考  Telnet两个标签页登录linux服务器，会话期示意图应该如何画？  [wbm01@localhost 04wait]$ who  root :0 2014-06-25 11:50  root pts/1 2014-06-25 11:52 (:0.0)  root pts/17 2014-06-25 22:10 (192.168.6.253)  root pts/18 2014-06-25 22:20 (192.168.6.253)  root pts/10 2014-06-25 13:17 (:0.0)  root pts/4 2014-06-25 16:49 (:0.0)  root pts/5 2014-06-25 19:47 (:0.0)  root pts/16 2014-06-25 20:18 (:0.0) |
| int setup\_daemon(int nochdir, int noclose);  int main(int argc, char \*argv[])  {  mydaemon(1, 1);  //man daemon 可以看到  //0表示改变重定向 1表示不改变  //daemon(1, 1);  printf("test ...\n");  for (;;) ;  return 0;  }  int mydaemon(int nochdir, int noclose)  {  pid\_t pid;  pid = fork();  if (pid == -1)  ERR\_EXIT("fork error");  if (pid > 0)  exit(EXIT\_SUCCESS);  setsid();    if (nochdir == 0)  chdir("/");  if (noclose == 0)  {  int i;  for (i=0; i<3; ++i)  close(i);  //相当于把0号文件描述符之下/dev/null  open("/dev/null", O\_RDWR); //fd文件描述符fd-0的文件描述符指向 -16  dup(0); //把0号文件描述符 赋值给空闲的文件描述符 1  dup(0); //把0号文件描述符 赋值给空闲的文件描述符 2  }  return 0;  } |
|  |
|  |
|  |

# 进程项目开发进价

## 多进程测试框架搭建

## 多进程遇上动态库