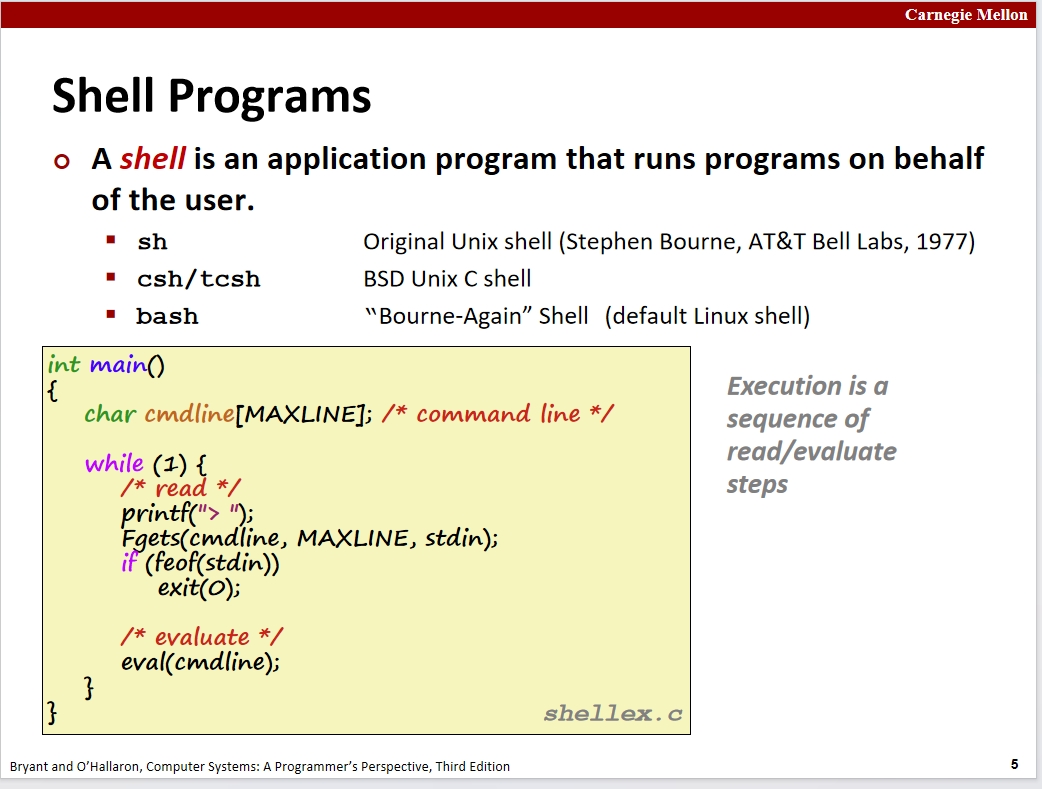
对于shell程序的一些分析

首先需要知道的是，一个shell语句其实本质上可以看做一个程序。

只是这个程序是已经定义好的了，你只需要根据特定的规则进行使用即可。在linux上基础的shell为bash，当然还有一些其他的版本。

对于一个shell，看起来就像是在运行一个程序

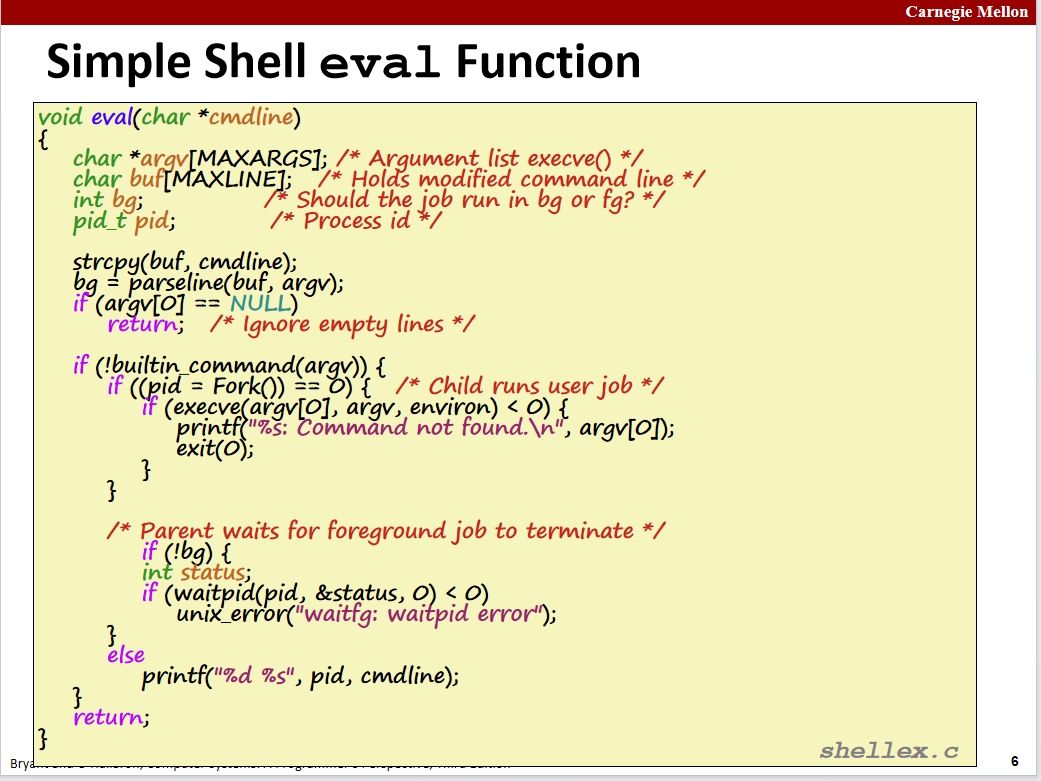


当shell运行时，其会在终端上打印一个>，然后等待用户输入，这个会像其他程序一样被压入输入流中然后等待处理。通常对于用户来说，shell的第一个字符是一个命令，接下来再输入的参数会是一系列的参数，就比如我们的gcc编译

gcc myproject

当我们输入字符并敲下回车之后，shell会检测这段输入流的末尾，对于终端来说这通常是ctrl+d,这个的意义就跟EOF一样，然后终端就会进行这段程序的运行。

至于一系列的命令行指令，可以把这些理解为高级语言中的语法，而且这些语法是十分的简洁且高效的



这个程序可以视作是一个c程序对shell程序对我们输入的信息的操作的模拟。当输入后，程序会对第一个参数进行解析并判断它是否是一个内置的shell命令，如果是，那么这个shell程序会直接执行。如果不是，这个shell会fork一个子进程，并在这个子进程中会使用execve实现对于命令行程序的进一步解析。将我们输入的命令行中第一个参数作为调用execve函数的第一个参数，然后把接下来的命令行参数和其他的一些环境设置传递给子进程。

在此，我们可以理解正确提到的一些概念了，为什么main函数会被传递参数，就是在这，也就是说程序运行前还有的启动程序就是一段自动生成的shell脚本。这个脚本通过程序的环境，配置等信息自动生成，并在我们想要点击程序运行时先自动运行，这个会在命令行中启动shell脚本进行启动，接下来会传递对应的命令行参数和环境变量，如果main函数需要接收的话。对于系统来说，无论是shell脚本还是将要执行的代码的可执行文件，这些都需要加载到内存中才能够执行，因此这也是正确main函数启动之前还存在一个栈帧的原因

这里有一个点差点卡到:  
1~系统对于子进程的内存分配。首先，要明确这个分配的空间是不一定会跟父进程的内存空间大小一致的。

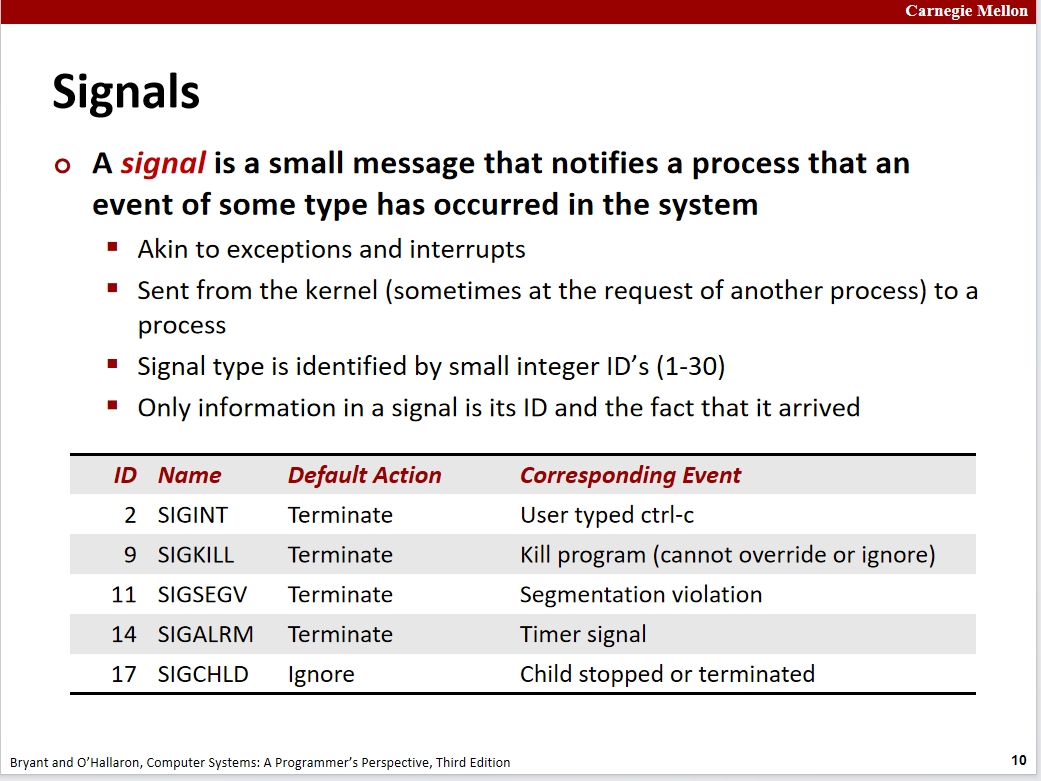
2~系统采取写时分配的操作，也就是说，子进程的创建在创建后并不会第一时间就将对应的数据复制过去，而是等待。此时，父子进程还是共用一个物理内存页，此时俩者还是共用一个内存空间。只有当系统检测到子进程想要对父进程的内存数据进行修改时，系统才会给子进程分配这些数据需要的进程空间，将这些数据复制过去，并将想要进行的操作进行执行。也就是说，内存的分配不是一开始就进行执行的。而且，这次分配后的内存将会是子进程的整个运行阶段的内存。因为当涉及到了想要进行数据的更改的阶段时，系统一定已经解析完了整个程序的架构，并计算出了这个程序需要分配多少内存。此时再进行分配内存能够提高效率

这里又差点给卡到，对于进程的内存分配，这里又涉及到的编译器的操作了，虽然内存分配这一块是由操作系统负责的，但是操作系统也要根据编译器给出的代码来识别当前进程应该分配哪些，多少内存空间。通过这些操作，操作系统能够保证内存的利用率。

当我们编写的代码逻辑很混乱时，编译器将无法优化代码，同时可能导致部分进程的代码会转移到我们不期望的进程中去，这个可能导致逻辑上的错误。更直接的，由于操作系统是根据代码的内容等分配内存的，在这种情况下，对于可能的内存错误，这个是很致命的

**信号，启动！！！！！**

信号可以通知进程系统中发生了一次某种类型的时间。这个与之前的异常相像，但是它完全是由软件实现的。对于内核信号，它完全是由内核发出的。但是，有时会在某些进程中通信中使用，有时某些进程需要给其他进程发送信息。对于大部分信号，它跟异常一样，也有一个对应的唯一的id。几个举例如下图

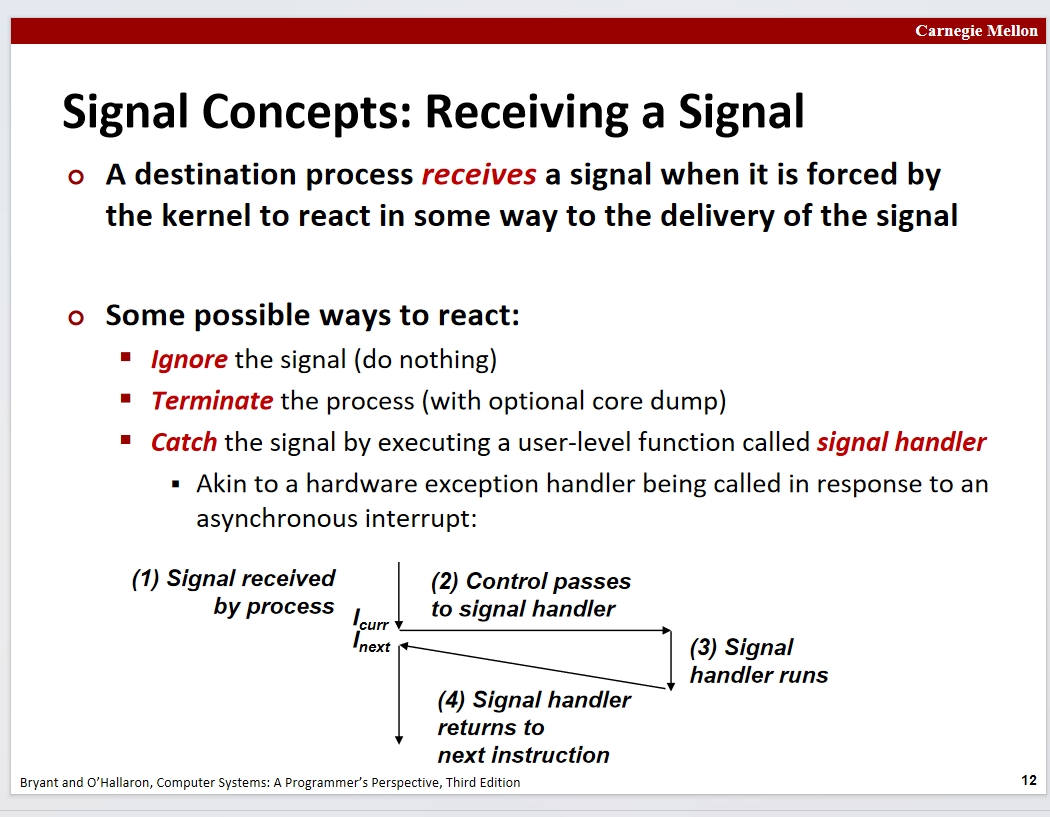


信号更多的是一种触发的机制。就比如当你访问一个不属于你的内存空间时，由于内存空间这一块是由操作系统管理的，故这里会触发操作系统向执行这个操作的进程发送SIGSEGV，中止当前进程并提示这个进程的这一步操作出现了段错误

看到最后一个SIGCHILD信号，这个信号在子进程被杀死时别发出，用于提醒父进程进行对应的操作(如果有的话)，这个也是shell对于那些可能的僵尸进程的处理

对于信号产生的可能情况：  
1~一个进程触发了内核的某种机制导致内核发送信号给触发的进程

2~一个进程需要进行一定的进程间通信，但是无法直接联系上，只能像电话一样通过内核进行联系



对于进程接收到信号后可能执行的事件：  
 1~ignore 即不做任何处理

2~terminate 中止当前进程

3~catch 通过用户自定义的一些信号处理函数进程处理

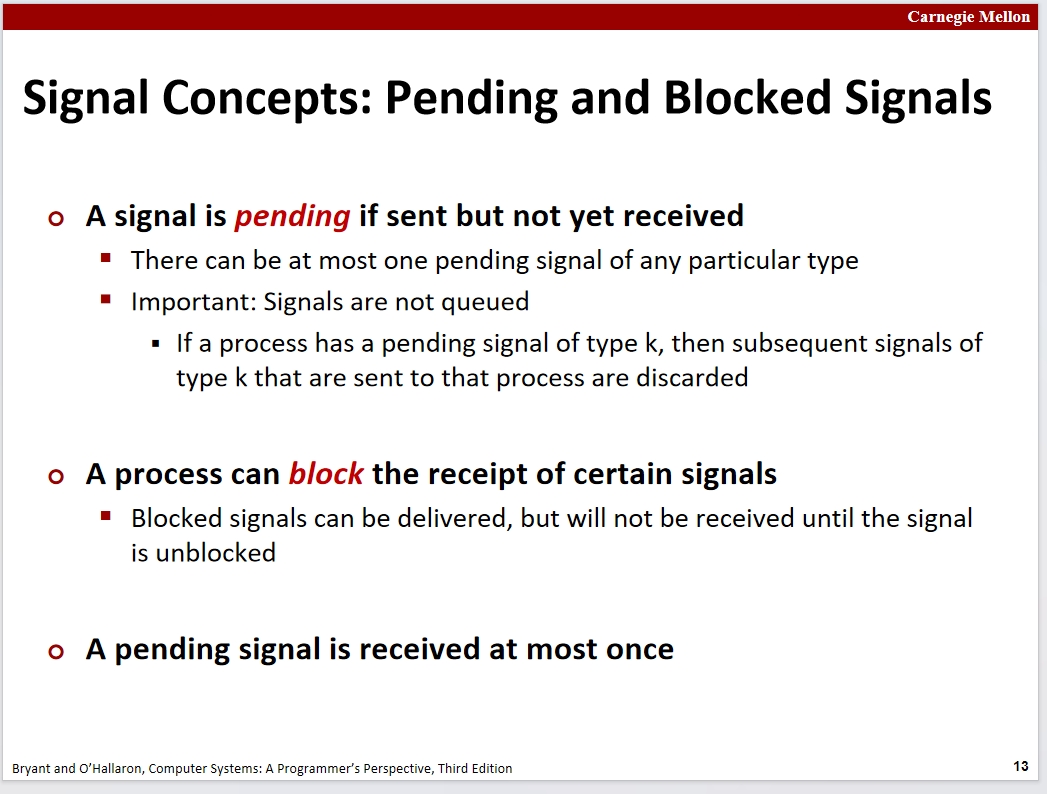
从这里也可以看出异常处理跟信号处理间的区别了，对于异常处理，都是内核级别的，而这里的信号处理一般都是用户级别的，即一般都是程序能自定义的。这个其实跟QT很像，不是吗

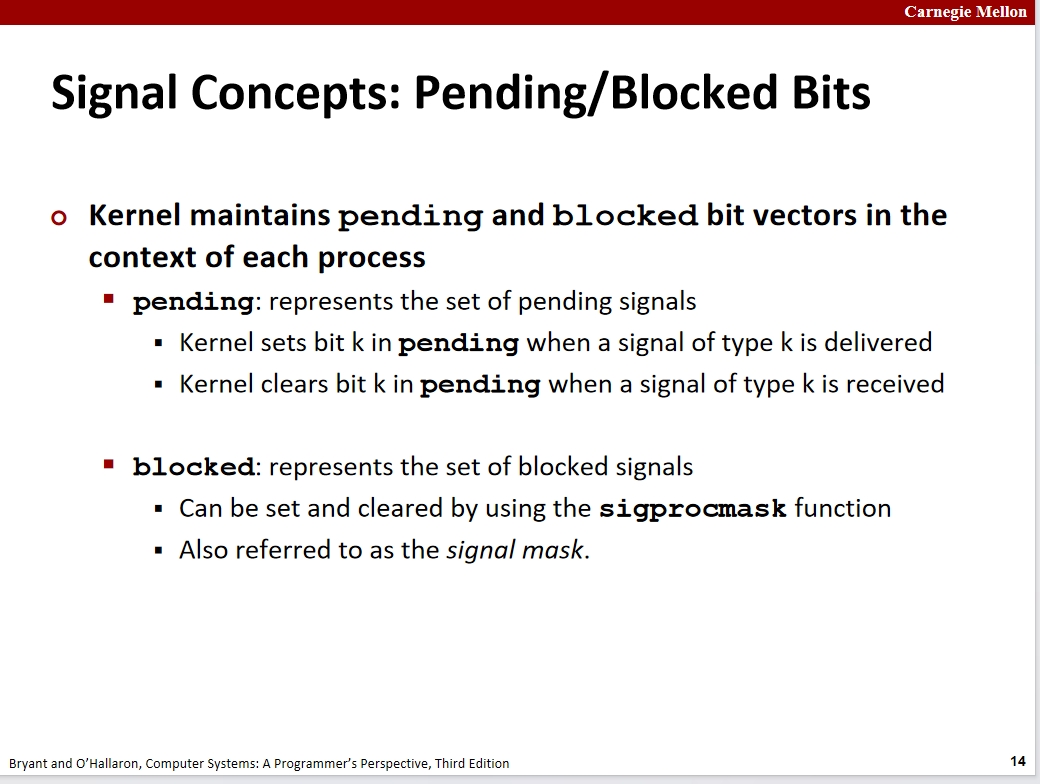
信号几个需要注意的点:

1~对于任何一个时候，每种信号都只能有一个正在被发送或准备被发送。对于信号，是不存在排队的做法的，因为每个时候就只能有一个，到后面更好理解

2~进程可以阻塞部分信号的接收，但是需要注意，进程无法阻止信号的发送，它的阻塞只能阻止进程在接受到信号后的处理，故被阻塞的信号可以被发送，但是在阻塞结束前它不会被接收

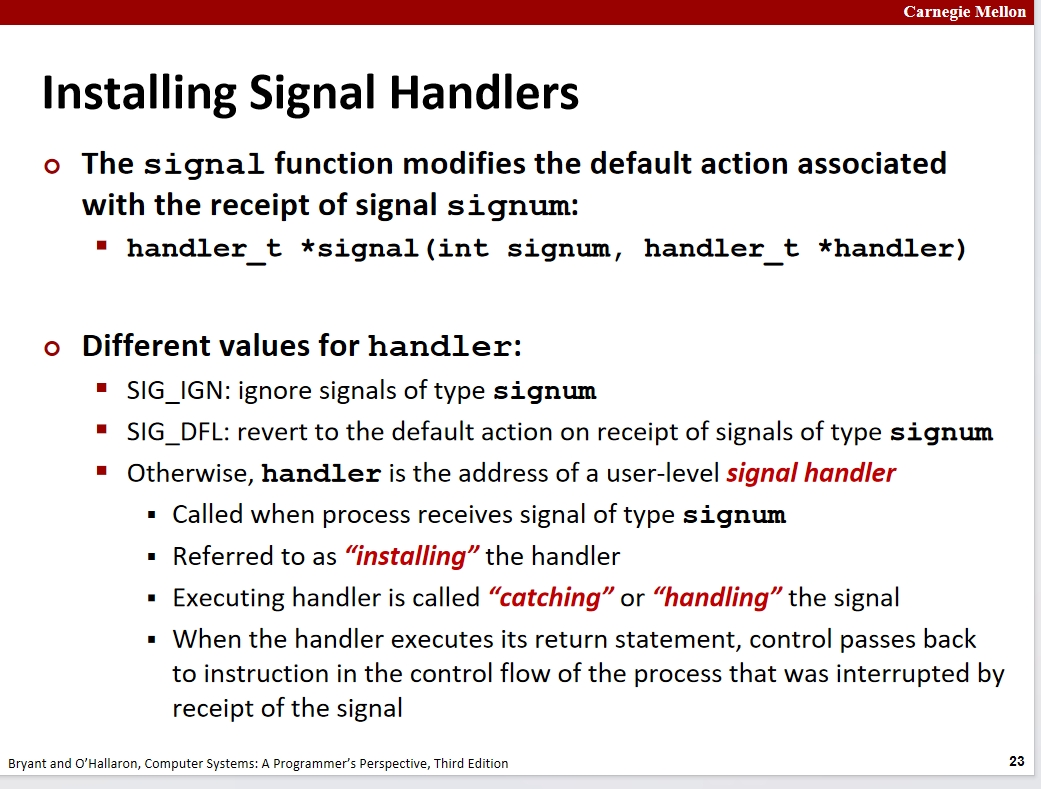
3~对于一个待处理信号，最多处理一次



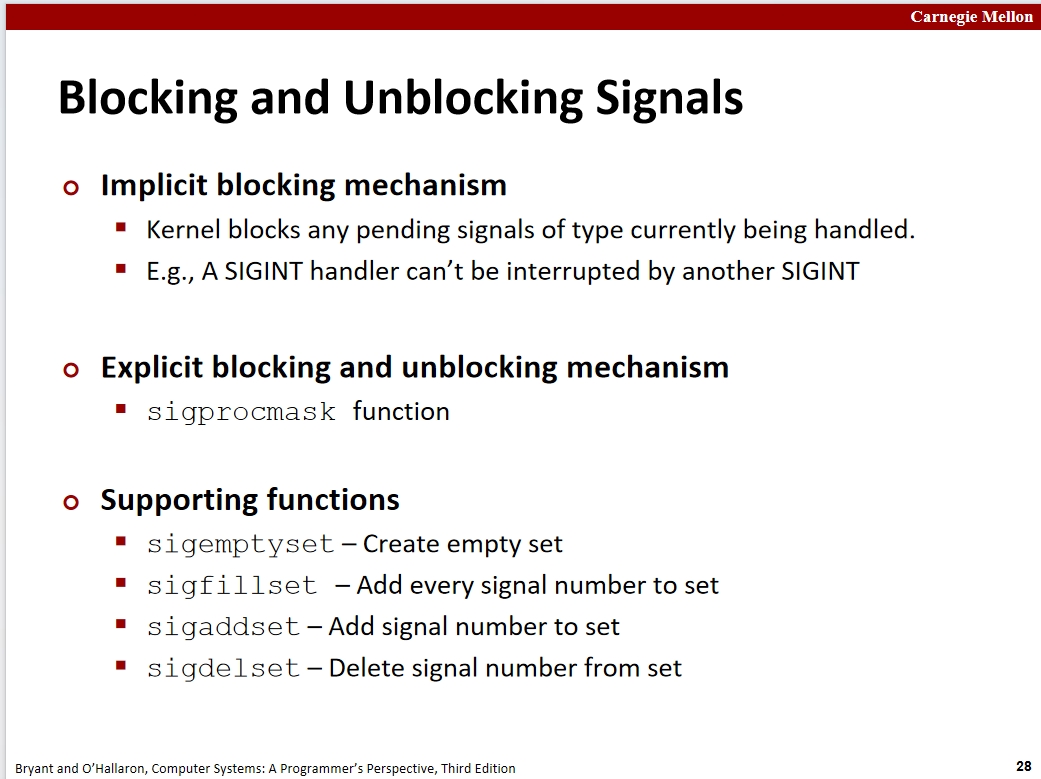


接下来分析下为什么，对于所有的信号，都只是位于一个特殊的信号向量上的一个位，注意是一个位，这上面一个位就对应着一个信号，俩个类型。当我们传递信号时，内核只是去设置对应的位上的数字，当我们需要传递同类型的另一个信号时，它只是会把这个位上的数字重新设置而已。当信号传递到内核时，内核会将这个位的状态设置为待处理，并且当信号被接收时程序将该位初始化。

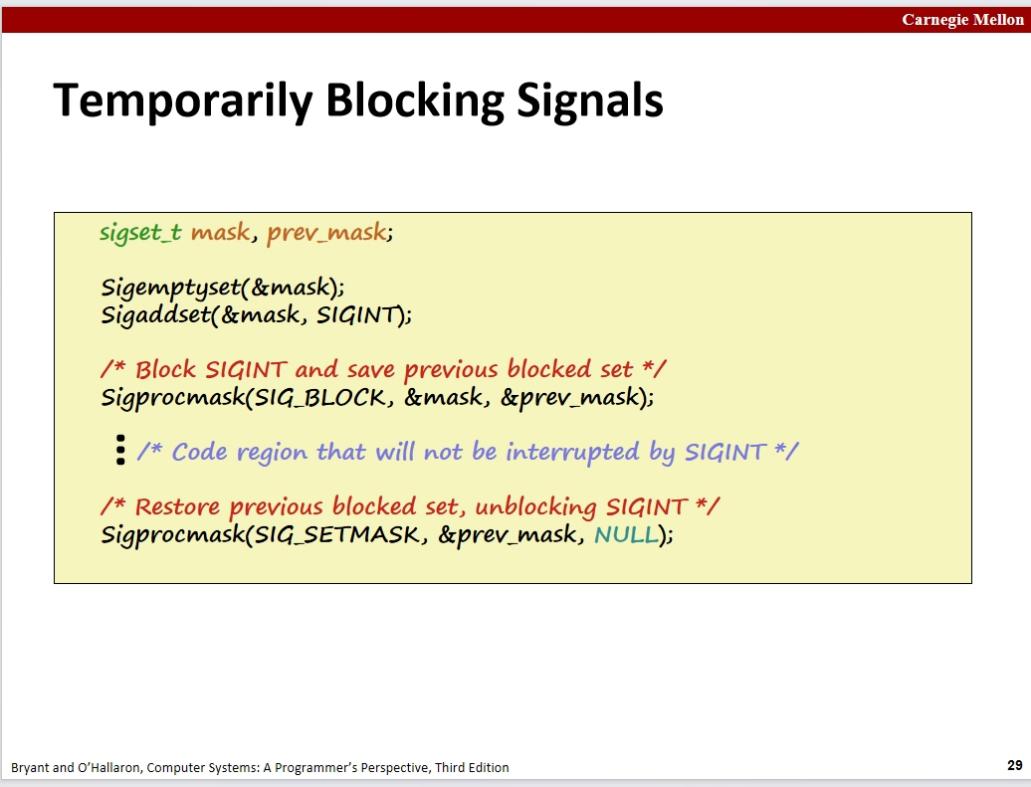
因此，内核中存在着对应着32为信号向量位的阻塞向量，这俩都是一个32位的int类型。



接下来看一个与connect函数很像的函数signal，signal函数可以关联对应的信号编号与一些信号处理。这个信号处理可以是宏，就比如SIG\_IGN或者SIG\_DFL。也可以是一个用户自定义的信号处理函数，对于这个函数，当信号被进程接收时，会发生中断，此时程序进程的事情将会被暂停，并优先进行这个函数的处理，在这个信号处理函数处理完后，再放回这个接受信号的语句的下一句继续执行，这跟QT很像吧。



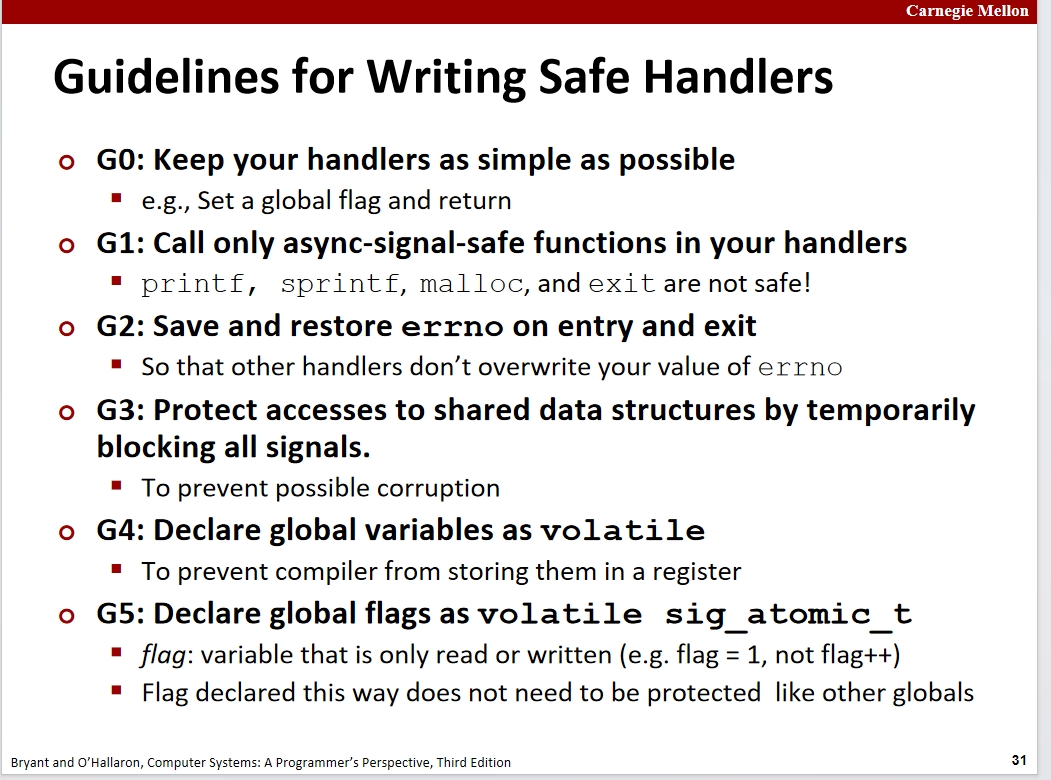
接下来看一个可以显示的设置信号阻塞和取消阻塞的函数sigprocmask，这个函数是一个内核的系统调用。这类函数允许一些功能，包括但不限于创建一个空的位向量，这个对应着需要的信号。对这些对应信号位上的数据进行置位以达到想要的效果等。



接下来看一下这里的使用例子，首先可以通过一个对应的mask掩码参数通过sigemptyset创建一个空的向量集合。正如之前所说，内核会在发射信号时检查对应的阻塞位，如果被阻塞了，那么这个信号最终将不会被接收，只可能被发送。

此时集合中没有元素，接下来通过Sigaddset函数将这个集合和对应的想要进行处理的信号关联起来。

接下来我们想要阻塞这个信号的话，需要用到Sigprocmask函数，第一个参数是想要进行处理的操作，可以是阻塞宏SIG\_BLOCK，也可以是其他的宏，不能是自定义函数，第二个将这个信号关联上的掩码，这个参数保存了这个信号的状态，是阻塞还是什么。第三个是我们想要对这个掩码进行更新设置复制的数据，其实也是一个掩码，只是这个参数最终会保存调用这个函数前第二个参数的状态，这个一般可以用于对于状态的恢复。

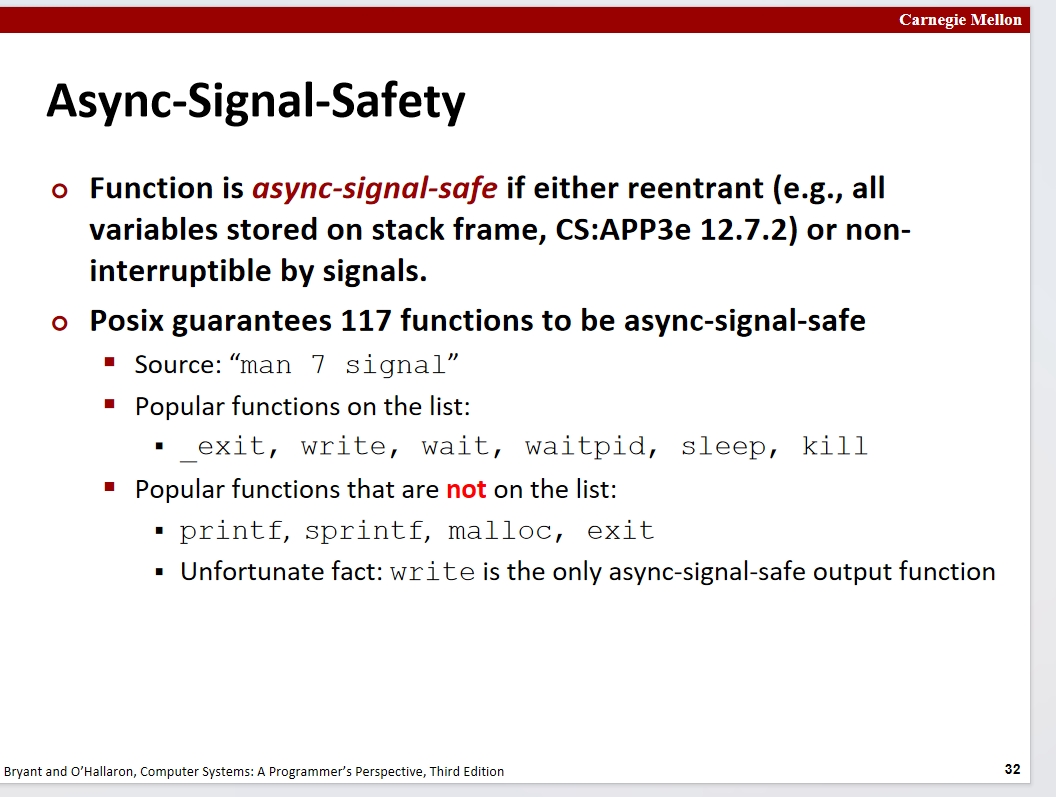
首先，我们需要简单了解一下这里设置的原理。对于信号状态的设置，只有底层的掩码才能进行更改。因此，这里在修改我们这里的sigset\_t变量的同时内核也会修改对应的掩码位。这里前面的Sigaddset函数会将对应的信号与我们定义的mask关联起来，其实应该也是追踪了每个位的使用情况。  


几种编写安全代码的规则，这里简单分析下G5

这里说对于想要追踪信号是否处理的全局变量，一般都得使用volatile关键字，这个关键字会修改这个全局变量的一些标志位，使得编译器将不会对其进行优化。否则，在程序中，编译器如果使用高级别优化，可能会将这个涉及到全局变量的检测给优化掉导致出现一些不符合我们想要的逻辑的事情。当不使用这个关键字时，编译器可能将这个全局变量放于寄存器中，甚至于将其视作一个常量，即使程序中对其进行了修改，底层上程序对其还是没有进行修改。

简单来说，这个关键字在这里会用到的效果就是告诉编译器，这个变量是会可能由于一些外部因素（如中断、硬件事件、其他线程等）导致修改的，编译器将不会对涉及到这个变量的任何条件判断进行优化

同理，还有一个可以用来表示变量不会被修改，只能被读写的关键字sig\_atomic\_t

接下来进入到一个信号安全的模块

首先如上图中的Async-Signal-Sagety，对于信号函数中，我们也有一些安全规范的函数，至于为什么，原因有几个。

1~首先要注意带信号函数的触发条件，这个函数是高于当前执行的程序的，由内核直接控制，当程序触发一个信号时，就比如计时器触发，或者在控制台中输入ctrl+c时，都会向内核中发射信号，然后内核会中断当前程序控制流，并执行对应的信号函数中的程序。

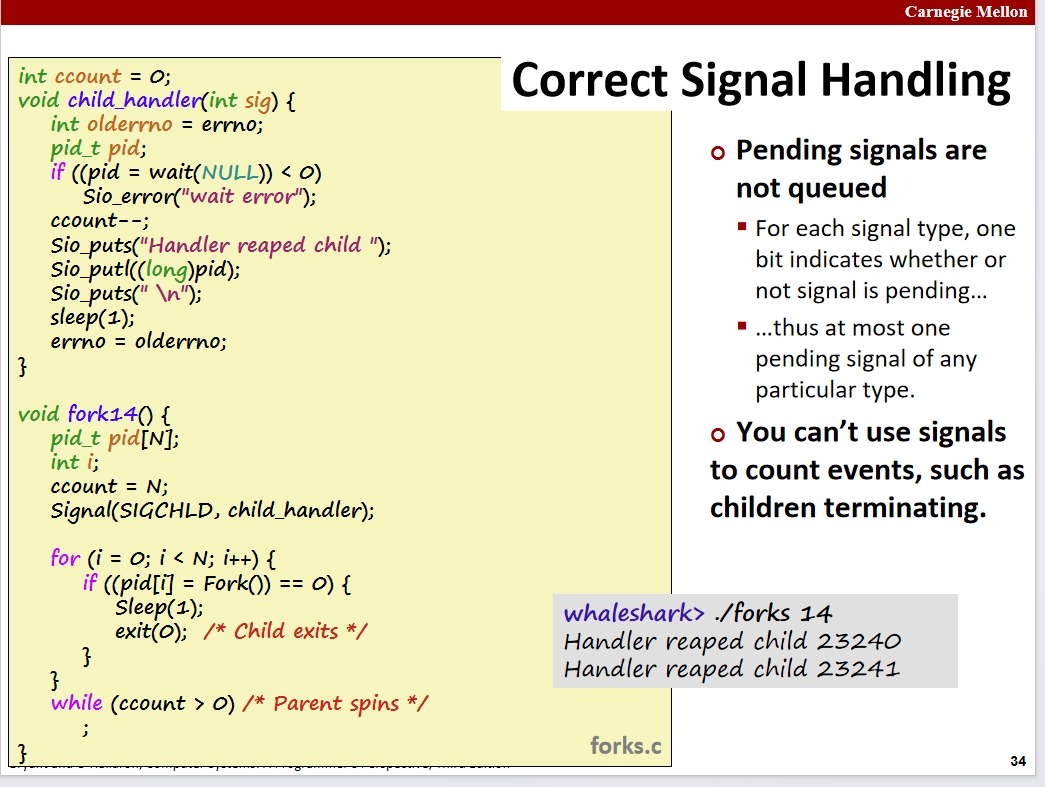
2~对于整个信号函数，函数是有修改和访问整个进程数据的能力的。因此，这就可能导致数据上的冲突。就比如，当我们在一个malloc分配内存时触发一个信号，程序将会在分配这个内存前跳转到信号函数流的执行，如果在这个函数中还有对于内存的申请，那么对于申请的位置，将会是与前面说的malloc函数的起始位置一致的，当我们的信号函数中没有对该函数中申请的内存进行合适的管理时，就比如这个分配内存会留存在栈中。函数执行完后回到信号发出地，程序会执行malloc函数。但是，程序此时会发现想要使用的地方的内存已经被只要，或者说，内核此时会检测到程序尝试写入一个已经使用的内存块，这将导致错误，甚至于程序崩溃。

因此，在信号函数中保持规范是相当重要的，这里如上图中的

\_exit, write, wait, waitpid, sleep, kill

能够保证在信号函数中正常使用且不会影响程序的正常功能，相反，对于一些函数printf, sprintf, malloc, exit 这些函数是可能在使用过程中出现问题的，所以这里对于这种不安全的函数得谨慎使用。

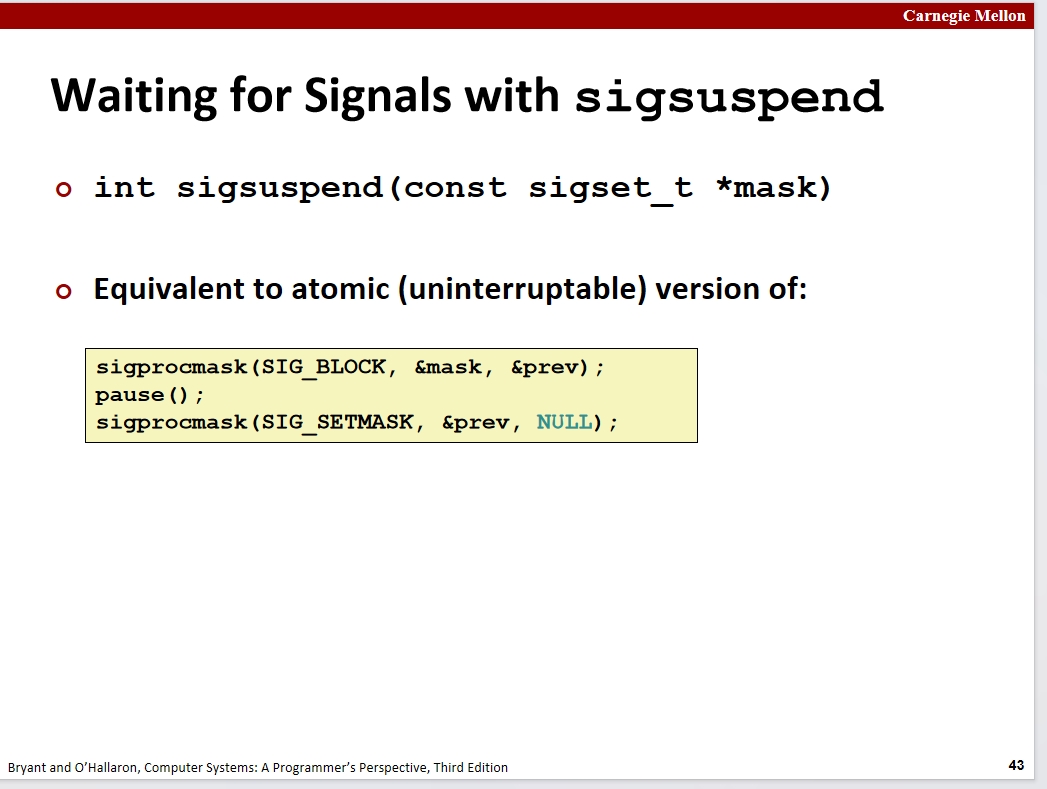
当我们在printf函数使用中触发了信号调用了一个也使用printf函数的信号函数时，这个会由于printf函数中的锁机制导致一个死锁的出现，导致程序会一直等待main函数中锁的释放，但是这个是永远释放不了的，所以这里会出现一个局部死锁的情况



接下来分析下不能使用信号函数来进行信号发送的计数的原因。 首先，信号的触发是一对一的，就比如当一个进程被结束时，这就会触发一个sigchild信号。内核会接受所有这些信号，但是内核发射回父进程的信号不一定会与这一系列信号的数量相一致，当我们触发的信号种类一致并且是连续的时候，内核会对这些信号进行合并，只会发送回一个对应的信号，也只触发一次信号函数。也就是说，对于信号的发出和信号的接收，这俩者是不一定相同的。

还有，我们也不能通过一个连续的子进程的关闭来确定就内核就只会发送回一次信号。就比如，在这些函数的调用过程中，还可能有雨进程的计时器到期导致又给内核发送了一个信号，而这个信号与之前的信号不同，导致这个信号流被分割了开来，导致这里的进程信号调用次数将无法预测

接下来看一种在一些环境中会经常使用的代码结构



在这之前，我们需要了解一个原子操作的概念。对于程序中的原子操作，这通常意味着这是一个无法被其他操作打断的，在执行这一个操作的过程中，它会位于所有的操作的优先级之上直到这个操作被完成。这里的sigprocmask就是一个原子操作，这样就意味着这个操作不会被任何的信号所打断。在这个函数执行期间所触发的所有信号将不会被内核所发送，将会位于一个内核的消息队列中。

接下来分析下这里的三行代码的逻辑。这里，首先，通过sigprocmask进行一段阻塞，通过这个操作使得在这个阶段使得所有想在这个阶段内核发送给进程被阻塞知道这个原子操作完成，并且在完成后使得那些应该被发送且没有被阻塞的信号发送给进程。

接下来，会调用pause函数，知道进程再次收到信号。这个过程其实跟前面的  
 while(!pid) pause();

一样，但是，这里的最大的优化就是我们这里的三行代码大大优化了CPU的占用率，因为在这个过程中CPU只是去等待是否有信号而不用去一直检查pid的状态。还有一系列的优点在这里不进行展开。

接下来考虑对pause()函数的退出，其实分析后我们可以看出，对于在这段代码中，我们前面这个原子操作导致暂缓发送到进程的信号是不会等到在pause函数执行时才使用的，相反，这些信号会在这个原子函数和pause操作之间被使用。因此，想要使得pause函数能够正确的退出，就必须保证在pause期间能够有信号能够被进程所接收。这就意味着这种结构也必须保证在pause期间能够有信号被发送并被进程接收。因此，所有的信号不能全部被屏蔽，而且必须保证在pause过程中程序一定得有发射并接受信号的行为。这里最简单的就是保证计时器信号不被阻塞且不会链接上奇奇怪怪的信号函数。

再下来考虑下这种结构中的扩展，对于这种结构，我们可以很明显的看到在pause之前，信号设置状态前这一段时间是一段相对特殊的时间。在这段时间内，有之前积压的信号等待处理，且这一段时间会屏蔽一些信号(上面的原子函数决定的)。因此，在这一段时间内我们可以添加一些处理代码，这些代码应该是对信号敏感的，一些期望之外的信号会由于对应的信号函数而改变这些代码的可能操作导致错误等。总而言之，这一个代码块，起到了一定的信号屏蔽的作用，能够用于一些特殊的功能