What Were They Thinking?

Annoyances Caused by Unsafe Assumptions

---

skape
mmiller@hick.org

Last modified: 04/04/2005

Contents

Chapter 1

Introduction

There is perhaps no issue more dear to a developer's heart than the issue of interoperability with third-party applications. In some cases, software that is being written by one developer has to be altered in order to make it function properly when used in conjunction with another application that is created by a third-party. For the sake of illustration, the lone developer will henceforth be referred to as the protagonist given his or her valiant efforts in their quest to obtain that which is almost always unattainable: interoperability. The third-parties, on the other hand, will be referred to as the antagonists due to their wretched attempts to prevent the protagonist from obtaining his or her goal of a utopian software environment. Now, granted, that's not to say that the protagonist can't also become the antagonist by continuing the ugly cycle of exposing compatibility issues to other would-be protagonists, but for the sake of discussion such a point is not relevant.

What is relevant, however, are the ways in which an antagonistic developer can write software that will force other developers to work around issues exposed by the software that the antagonist has written. There are far too many specific issues to list, but the majority of these issues can be generalized into one category that will serve as the focus for this document. To put it simply, many developers make assumptions about the state of the machine that their software will be executing on. For instance, some software will assume that they are the only piece of software performing a given task on a machine. In the event that another piece of software attempts to perform a similar task, such as may occur when two applications need to extend APIs by hooking them, the results may be unpredictable. Perhaps a more concrete example of where assumptions can lead to problems can be seen when developers assume that the behavior of undocumented or unexposed APIs will not change.

Before putting all of the blame on the antagonists, however, it is important to

understand that it is, in most cases, necessary to make assumptions about the way in which undocumented code performs, such as when dealing with low-level software. This is especially true when dealing with closed-source APIs, such as those provided by Microsoft. To that point, Microsoft has made an effort to document the ways in which every exposed API routine can perform, thereby reducing the number of compatibility issues that a developer might experience if they were to assume that a given routine would always perform in the same manner. Furthermore, Microsoft is renowned for attempting to always provide backwards compatibility. If a Microsoft application performs one way in a given release, chances are that it will continue to perform in the same fashion in subsequent releases. Third-party vendors, on the other hand, tend to have a more egocentric view of the way in which their software should work. This leads most vendors to dodge responsibility by pointing the blame at the application that is attempting to perform a certain task rather than making their code to be more robust.

In the interest of helping to make code more robust, this document will provide two examples of widely used software that make assumptions about the way in which code will execute on a given machine. The assumptions these applications make are always safe under normal conditions. However, if a new application that performs a certain task or an undocumented change is thrown into the mix, the applications find themselves faltering in the most unenjoyable ways. The two applications that will be analyzed are listed below:

1. McAfee VirusScan Consumer (8.0/9.0)
2. ATI Radeon 9000 Driver Series

Each of the assumptions that these three software products make will be analyzed in-depth to describe why it is that they are poor assumptions to make, such as by describing or illustrating conditions where the assumptions are, or could be, false. From there, suggestions will be made on how these assumptions might be worked around or fixed to allow for a more stable product in general. In the end, the reader should have a clear understanding of the assumptions described in this document. If successful, the author hopes the topic will allow the reader to think critically about the various assumptions the reader might make when implementing software.

Chapter 2

McAfee VirusScan Consumer (8.0/9.0)

2.1 The Assumption

McAfee VirusScan Consumer 8.0, 9.0, and possibly previous versions make assumptions about processes not performing certain types of file operations during a critical phase of process initialization. If file operations are performed during this phase, the machine may blue screen due to an invalid pointer access.

2.2 The Problem

The critical phase of process execution that the summary refers to is the period between the time that the new process object instance is created by `nt!ObCreateObject` and the time the new process object is inserted into the process object type list by `nt!ObInsertObject`. The reason this phase is so critical is because it is not safe for things to attempt to obtain a handle to the process object, such as can be done by calling `nt!ObOpenObjectByPointer`. If an application were to attempt to obtain a handle to the process object before it had been inserted into the process object list by `nt!ObInsertObject`, critical creation state information that is stored in the process object's header would be overwritten with state information that is meant to be used after the process has passed the initial security validation phase that is handled by `nt!ObInsertObject`. In some cases, overwriting the creation state information prior to calling `nt!ObInsertObject` can lead to invalid pointer references when `nt!ObInsertObject` is eventually called, thus leading to an evil blue screen that some users are all too familiar

with.

To better understand this problem it is first necessary to understand the way in which `nt!PspCreateProcess` creates and initializes the process object and the process handle that is passed back to callers. The object creation portion is accomplished by making a call to `nt!ObCreateObject` in the following fashion:

```
ObCreateObject(
        KeGetPreviousMode(),
        PsProcessType,
        ObjectAttributes,
        KeGetPreviousMode(),
        0,
        0x258,
        0,
        0,
        &ProcessObject);
```

If the call is successful, a process object of the supplied size is created and initialized using the attributes supplied by the caller. In this case, the object is created using the `nt!PsProcessType` object type. The size argument that is supplied to `nt!ObCreateObject`, which in this case is `0x258`, will vary between various versions of Windows as new fields are added and removed from the opaque `EPROCESS` structure. The process object's instance, as with all objects, is prefixed with an `OBJECT_HEADER` that may or may not also be prefixed with optional object information. For reference, the `OBJECT_HEADER` structure is defined as follows:

```
OBJECT_HEADER:
   +0x000 PointerCount     : Int4B
   +0x004 HandleCount      : Int4B
   +0x004 NextToFree       : Ptr32 Void
   +0x008 Type             : Ptr32 _OBJECT_TYPE
   +0x00c NameInfoOffset   : UChar
   +0x00d HandleInfoOffset : UChar
   +0x00e QuotaInfoOffset  : UChar
   +0x00f Flags            : UChar
   +0x010 ObjectCreateInfo : Ptr32 _OBJECT_CREATE_INFORMATION
   +0x010 QuotaBlockCharged : Ptr32 Void
   +0x014 SecurityDescriptor : Ptr32 Void
   +0x018 Body             : _QUAD
```

When an object is first returned from `nt!ObCreateObject`, the `Flags` attribute will indicate if the `ObjectCreateInfo` attribute is pointing to valid data by

having the OB_FLAG_CREATE_INFO, or 0x1 bit, set. If the flag is set then the ObjectCreateInfo attribute will point to an OBJECT_CREATE_INFORMATION structure which has the following definition:

```
OBJECT_CREATE_INFORMATION:
   +0x000 Attributes       : Uint4B
   +0x004 RootDirectory    : Ptr32 Void
   +0x008 ParseContext     : Ptr32 Void
   +0x00c ProbeMode        : Char
   +0x010 PagedPoolCharge  : Uint4B
   +0x014 NonPagedPoolCharge : Uint4B
   +0x018 SecurityDescriptorCharge : Uint4B
   +0x01c SecurityDescriptor : Ptr32 Void
   +0x020 SecurityQos      : Ptr32 _SECURITY_QUALITY_OF_SERVICE
   +0x024 SecurityQualityOfService : _SECURITY_QUALITY_OF_SERVICE
```

When nt!ObInsertObject is finally called, it is assumed that the object still has the OB_FLAG_CREATE_INFO bit set. This will always be the case unless something has caused the bit to be cleared, as will be illustrated later in this chapter. The flow of execution within nt!ObInsertObject begins first by checking to see if the process' object header has any name information, which is conveyed by the NameInfoOffset of the OBJECT_HEADER. Regardless of whether or not the object has name information, the next step taken is to check to see if the object type that is associated with the object that is supplied to nt!ObInsertObject requires a security check to be performed. This requirement is conveyed through the TypeInfo attribute of the OBJECT_TYPE structure which is defined below:

```
OBJECT_TYPE:
   +0x000 Mutex            : _ERESOURCE
   +0x038 TypeList         : _LIST_ENTRY
   +0x040 Name             : _UNICODE_STRING
   +0x048 DefaultObject    : Ptr32 Void
   +0x04c Index            : Uint4B
   +0x050 TotalNumberOfObjects : Uint4B
   +0x054 TotalNumberOfHandles : Uint4B
   +0x058 HighWaterNumberOfObjects : Uint4B
   +0x05c HighWaterNumberOfHandles : Uint4B
   +0x060 TypeInfo         : _OBJECT_TYPE_INITIALIZER
   +0x0ac Key              : Uint4B
   +0x0b0 ObjectLocks      : [4] _ERESOURCE

OBJECT_TYPE_INITIALIZER:
   +0x000 Length           : Uint2B
   +0x002 UseDefaultObject : UChar
   +0x003 CaseInsensitive  : UChar
```

```
   +0x004 InvalidAttributes : Uint4B
   +0x008 GenericMapping    : _GENERIC_MAPPING
   +0x018 ValidAccessMask   : Uint4B
   +0x01c SecurityRequired : UChar
   +0x01d MaintainHandleCount : UChar
   +0x01e MaintainTypeList : UChar
   +0x020 PoolType          : _POOL_TYPE
   +0x024 DefaultPagedPoolCharge : Uint4B
   +0x028 DefaultNonPagedPoolCharge : Uint4B
   +0x02c DumpProcedure     : Ptr32
   +0x030 OpenProcedure     : Ptr32
   +0x034 CloseProcedure    : Ptr32
   +0x038 DeleteProcedure   : Ptr32
   +0x03c ParseProcedure    : Ptr32
   +0x040 SecurityProcedure : Ptr32
   +0x044 QueryNameProcedure : Ptr32
   +0x048 OkayToCloseProcedure : Ptr32
```

The specific boolean field that is checked by `nt!ObInsertObject` is the `TypeInfo.SecurityRequired` flag. If the flag is set to `TRUE`, which it is for the `nt!PsProcessType` object type, then `nt!ObInsertObject` uses the access state that is passed in as the second argument or creates a temporary access state that it uses to validate the access mask that is supplied as the third argument to `nt!ObInsertObject`. Prior to validating the access state, however, the `SecurityDescriptor` attribute of the `ACCESS_STATE` structure is set to the `SecurityDescriptor` of the `OBJECT_CREATE_INFORMATION` structure. This is done without any checks to ensure that the `OB_FLAG_CREATE_INFO` flag is still set in the object's header, thus making it potentially dangerous if the flag has been cleared and the union'd attribute no longer points to creation information.

In order to validate the access mask, `nt!ObInsertObject` calls into `nt!ObpValidateAccessMask` with the initialized `ACCESS_STATE` as the only argument. This function first checks to see if the `ACCESS_STATE`'s `SecurityDescriptor` attribute is set to NULL. If it's not, then the function checks to see if the `SecurityDescriptor`'s `Control` attribute has a flag set. It is at this point that the problem is realized under conditions where the object's `ObjectCreateInfo` attribute no longer points to creation information. When such a condition occurs, the `SecurityDescriptor` attribute that is referenced relative to the `ObjectCreateInfo` attribute will potentially point to invalid memory. This can then lead to an access violation when attempting to reference the `SecurityDescriptor` that is passed as part of the `ACCESS_STATE` instance to `nt!ObpValidateAccessMask`. For reference, the `ACCESS_STATE` structure is defined below:

```
ACCESS_STATE:
   +0x000 OperationID      : _LUID
```

```
   +0x008 SecurityEvaluated : UChar
   +0x009 GenerateAudit    : UChar
   +0x00a GenerateOnClose  : UChar
   +0x00b PrivilegesAllocated : UChar
   +0x00c Flags            : Uint4B
   +0x010 RemainingDesiredAccess : Uint4B
   +0x014 PreviouslyGrantedAccess : Uint4B
   +0x018 OriginalDesiredAccess : Uint4B
   +0x01c SubjectSecurityContext : _SECURITY_SUBJECT_CONTEXT
   +0x02c SecurityDescriptor : Ptr32 Void
   +0x030 AuxData          : Ptr32 Void
   +0x034 Privileges       : __unnamed
   +0x060 AuditPrivileges  : UChar
   +0x064 ObjectName       : _UNICODE_STRING
   +0x06c ObjectTypeName   : _UNICODE_STRING
```

Under normal conditions, `nt!ObInsertObject` is the first routine to create a handle to the newly created object instance. When the handle is created, the creation information that was initialized during the instantiation of the object is used for such things as validating access, as described above. Once the creation information is used it is discarded and replaced with other information that is specific to the type of the object being inserted. In the case of process objects, the `Flags` attribute has the `OB_FLAG_CREATE_INFO` bit cleared and the `QuotaBlockCharged` attribute, which is union'd with the `ObjectCreateInfo` attribute, is set to an instance of an `EPROCESS_QUOTA_BLOCK` which is defined below:

```
EPROCESS_QUOTA_ENTRY:
   +0x000 Usage            : Uint4B
   +0x004 Limit            : Uint4B
   +0x008 Peak             : Uint4B
   +0x00c Return           : Uint4B

EPROCESS_QUOTA_BLOCK:
   +0x000 QuotaEntry       : [3] _EPROCESS_QUOTA_ENTRY
   +0x030 QuotaList        : _LIST_ENTRY
   +0x038 ReferenceCount   : Uint4B
   +0x03c ProcessCount     : Uint4B
```

The assumptions made by `nt!ObInsertObject` work flawlessly so long as it is the first routine to create a handle to the object instance. Fortunately, under normal circumstances, `nt!ObInsertObject` is always the first routine to create a handle to the object. Unfortunately for McAfee, however, they assume that they can safely attempt to obtain a handle to a process object without first checking to see what state of execution the process is in, such as by checking to see if the

OB_FLAG_CREATE_INFO flag is set in the object's header. By attempting to obtain a handle to the process object before it is inserted by nt!ObInsertObject, McAfee effectively destroys state that is needed by nt!ObInsertObject to succeed.

To show this problem being experienced in the real world, the following debugger output shows McAfee first attempting to obtain a handle to the process object which is then followed shortly thereafter by nt!ObInsertObject attempting to validate the object's access mask with a bogus SecurityDescriptor which, in turn, results in an unrecoverable access violation:

```
McAfee attempting to open a handle to the process object before
nt!ObInsertObject has been called:

kd> k
nt!ObpChargeQuotaForObject+0x2f
nt!ObpIncrementHandleCount+0x70
nt!ObpCreateHandle+0x17c
nt!ObOpenObjectByPointer+0x97
WARNING: Stack unwind information not available.
NaiFiltr+0x2e45
NaiFiltr+0x3bb2
NaiFiltr+0x4217
nt!ObpLookupObjectName+0x56a
nt!ObOpenObjectByName+0xe9
nt!IopCreateFile+0x407
nt!IoCreateFile+0x36
nt!NtOpenFile+0x25
nt!KiSystemService+0xc4
nt!ZwOpenFile+0x11
0x80a367b5
nt!PspCreateProcess+0x326
nt!NtCreateProcessEx+0x7e
nt!KiSystemService+0xc4

After which point nt!ObInsertObject attempts to validate the
object's access mask using an invalid SecurityDescriptor:

kd> k
nt!ObpValidateAccessMask+0xb
nt!ObInsertObject+0x1c2
nt!PspCreateProcess+0x5dc
nt!NtCreateProcessEx+0x7e
nt!KiSystemService+0xc4
kd> r
eax=fa7bbb54 ebx=ffa9fc60 ecx=00023994
```

```
edx=00000000 esi=00000000 edi=ffb83f00
eip=8057828e esp=fa7bbb40 ebp=fa7bbbb8
iopl=0         nv up ei pl nz na pe nc
cs=0008  ss=0010  ds=0023  es=0023
fs=0030  gs=0000             efl=00000202
nt!ObpValidateAccessMask+0xb:
8057828e f6410210
    test    byte ptr [ecx+0x2],0x10 ds:0023:00023996=??
```

The method by which this issue was located was by setting a breakpoint on the instruction after the call to `nt!ObCreateObject` in `nt!PspCreateProcess`. Once hit, a memory access breakpoint was set on the `Flags` attribute of the object's header that would break whenever the field was written to. This, in turn, lead to the tracking down of the fact that McAfee was acquiring a handle to the process object prior to `nt!ObInsertObject` being called, which in turn lead to the `OB_FLAG_CREATE_INFO` flag being cleared and the `ObjectCreateInfo` attribute being invalidated.

2.3 The Solution

There are two ways that have been identified that could correct this issue. The first, and most plausible, would be for McAfee to modify their driver such that it will refuse to acquire a handle to a process object if the `OB_FLAG_CREATE_INFO` bit is set in the process' object header `Flags` attribute. The downside to using this approach is that it requires McAfee to make use of undocumented structures that are intended by Microsoft to be opaque, and for good reason. However, the author is not currently aware of another means by which an object's creation state can be detected using general purpose API routines.

The second approach, and it's one that should at least result in a bugcheck within `nt!ObInsertObject`, would be to check to see if the object's `OB_FLAG_CREATE_INFO` bit has been cleared. If it has, an alternate action can be taken to validate the object's access mask. If it hasn't, the current method of validating the access mask can be used. At this point in time, the author cannot currently speak on what the alternate action would be, though it seems plausible that there would be another means by which a synonymous action could be performed without relying on the creation information in the object header.

In the event that neither of these solutions are pursued, it will continue to be necessary for protagonistic developers to avoid performing actions between `nt!ObCreateObject` and `nt!ObInsertObject` that might result in file operations being performed from within the new process' context. One of a number of work-arounds to this problem would be to post file operations off to a system worker thread that would then inherently run within the context of the `System`

process rather than the new process.

Chapter 3

ATI Radeon 9000 Driver Series

3.1 The Assumption

The ATI Radeon 9000 Driver Series, and likely other ATI driver series, makes assumptions about the location that the `RTL_USER_PROCESS_PARAMETERS` structure will be mapped at in the address space of a process that attempts to do 3D operations. If the structure is not mapped at the address that is expected, the machine may blue screen depending on the values that exist at the memory location, if any.

3.2 The Problem

During some experimentation with changing the default address space layout of processes on NT-based versions of Windows, it was noticed that machines that were using the ATI Radeon 9000 series drivers would crash if a process attempted to do 3D operations and the location of the process' parameter information was changed from the address at which it is normally mapped at. Before proceeding, it is first necessary for the reader to understand the purpose of the process parameter information structure and how it is that it's mapped into the process' address space.

Most programmers are familiar with the API routine `kernel32!CreateProcess[A/W]`. This routine serves as the primary means by which user-mode applications spawn new processes. The function itself is robust enough to support a number of ways in which a new process can be initialized and then executed. Behind the

scenes, CreateProcess performs all of the necessary operations to prepare the new task for execution. These options include opening the executable image file and creating a section object that is then passed to `ntdll!NtCreateProcessEx` which returns a unique process handle on success. If a handle is obtained, CreateProcess then proceeds to prepare the process for execution by initializing the process' parameters as well as creating and initializing the first thread in the process. A more complete analysis of the way in which CreateProcess operates can be found in David Probert's excellent analysis of Windows NT's process architecture[1].

For the purpose of this document, however, the part that is of most concern is that step in which CreateProcess initializes the new process' parameters. This is accomplished by making a call into `kernel32!BasePushProcessParameters` which in turn calls into `ntdll!RtlCreateProcessParameters`. The parameters are initialized within the process that is calling CreateProcess and are then, in turn, copied into the address space of the new process by first allocating storage with `ntdll!NtAllocateVirtualMemory` and then by copying the memory from the parent process to the child with `ntdll!NtWriteVirtualMemory`. Due to the fact that this occurs before the new process actually executes any code, the address that the process parameter structure is allocated at is almost guaranteed to be at the same address. This address happens to be `0x00020000`. This fact is most likely why ATI made the assumption that the process parameter information would always be at a static address.

If, however, `ntdll!NtAllocateVirtualMemory` allocates the process parameter storage at any place other than the static address described above, ATI's driver will attempt to reference a potentially invalid address when it comes time to perform 3D operations. The specific portion of the driver suite that has the error is the `ATI3DUAG.DLL` kernel-mode graphics driver. Inside this image there is a portion of code that attempts to make reference to the addresses `0x00020038` and `0x0002003C` without doing any sort of probing and locking or validation on the region it's requesting. If the region does not exist or contains unexpected data, a blue screen is a sure thing. The actual portion of the driver that makes this assumption can be found below:

```
mov     [ebp+var_4], eax
mov     edx, 20000h                     <--
mov     [ebp+var_24], edx
movzx   ecx, word ptr ds:dword_20035+3 <--
shr     ecx, 1
mov     [ebp+var_28], ecx
lea     eax, [ecx-1]
mov     [ebp+var_1C], eax
test    eax, eax
jbe     short loc_227CC
mov     ebx, [edx+3Ch]                  <--
```

```
cmp     word ptr [ebx+eax*2], '\'
```

The lines of intereste are marked by "<--" indicators pointing to the exact instructions that result in a reference being made to an address that is expected to be within a process' parameter information structure. For the sake of investigation, one might wonder what it is that the driver could be attempting to reference. To determine that, it is first necessary to dump the format of the process parameter structure which, as stated previously, is RTL_USER_PROCESS_PARAMETERS:

```
RTL_USER_PROCESS_PARAMETERS:
   +0x000 MaximumLength    : Uint4B
   +0x004 Length           : Uint4B
   +0x008 Flags            : Uint4B
   +0x00c DebugFlags       : Uint4B
   +0x010 ConsoleHandle    : Ptr32 Void
   +0x014 ConsoleFlags     : Uint4B
   +0x018 StandardInput    : Ptr32 Void
   +0x01c StandardOutput   : Ptr32 Void
   +0x020 StandardError    : Ptr32 Void
   +0x024 CurrentDirectory : _CURDIR
   +0x030 DllPath          : _UNICODE_STRING
   +0x038 ImagePathName    : _UNICODE_STRING
   +0x040 CommandLine      : _UNICODE_STRING
   +0x048 Environment      : Ptr32 Void
   +0x04c StartingX        : Uint4B
   +0x050 StartingY        : Uint4B
   +0x054 CountX           : Uint4B
   +0x058 CountY           : Uint4B
   +0x05c CountCharsX      : Uint4B
   +0x060 CountCharsY      : Uint4B
   +0x064 FillAttribute    : Uint4B
   +0x068 WindowFlags      : Uint4B
   +0x06c ShowWindowFlags  : Uint4B
   +0x070 WindowTitle      : _UNICODE_STRING
   +0x078 DesktopInfo      : _UNICODE_STRING
   +0x080 ShellInfo        : _UNICODE_STRING
   +0x088 RuntimeData      : _UNICODE_STRING
   +0x090 CurrentDirectores : [32] _RTL_DRIVE_LETTER_CURDIR
```

To determine the attribute that the driver is attempting to reference, one must take the addresses and subtract them from the base address 0x00020000. This produces two offsets: 0x38 and 0x3c. Both of these offsets are within the ImagePathName attribute which is a UNICODE_STRING. The UNICODE_STRING structure is defined as:

```
UNICODE_STRING:
   +0x000 Length           : Uint2B
   +0x002 MaximumLength    : Uint2B
   +0x004 Buffer           : Ptr32 Uint2B
```

This would mean that the driver is attempting to reference the path name of the process' executable image. The `0x38` offset is the length of the image path name and the `0x3c` is the pointer to the image path name buffer that actually contains the path. The reason that the driver would need to get access to the executable path is outside of the scope of this discussion, but suffice to say that the method on which it is based is an assumption that may not always be safe to make, especially under conditions where the process' parameter information is not mapped at `0x00020000`.

3.3 The Solution

The solution to this problem would be for ATI to come up with an alternate means by which the process' image path name can be obtained. Possibilities for alternate methods include referencing the PEB to obtain the address of the process parameters (by using the `ProcessParameters` attribute of the PEB). This approach is suboptimal because it requires that ATI attempt to reference fields in a structure that is intended to be opaque and also readily changes between versions of Windows. Another alternate approach, which is perhaps the most feasible, would be to make use of the `ProcessImageFileName` PROCESSINFOCLASS. This information class can be queried using the `NtQueryInformationProcess` system call to populate a `UNICODE_STRING` that contains the full path to the image that is associated with the handle that is supplied to `NtQueryInformationProcess`. The nice thing about this is that it actually indirectly uses the alternate method from the first proposal, but it does so internally rather than forcing an external vendor to access fields of the PEB.

Regardless of the actual solution, it seems obvious that assuming that a region of memory will be mapped at a fixed address in every process is something that ATI should not do. There are indeed cases where Windows itself requires certain things to be mapped at the same address between one execution of a process to the next, but it is the opinion of the author that ATI should not assume things that Windows itself does not also assume.

Chapter 4

Conclusion

Though this document may appear as an attempt to make specific 3rd party vendors look bad, that is not its intention. In fact, the author acknowledges having been an antagonistic developer in the past. To that point, the author hopes that by providing specific illustrations of where assumptions made by 3rd parties can lead to problems, the reader will be more apt to consider potential conditions that might become problematic if other applications attempt to co-exist with ones that the reader may write in the future.

Bibliography

[1] Probert, David B. *Windows Kernel Internals: Process Architecture.* http://www.i.u-tokyo.ac.jp/ss/lecture/new-documents/Lectures/13-Processes/Processes.ppt; accessed April 04, 2005.