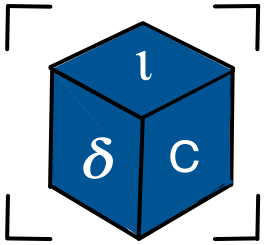
# Resource Virtualization with Containers

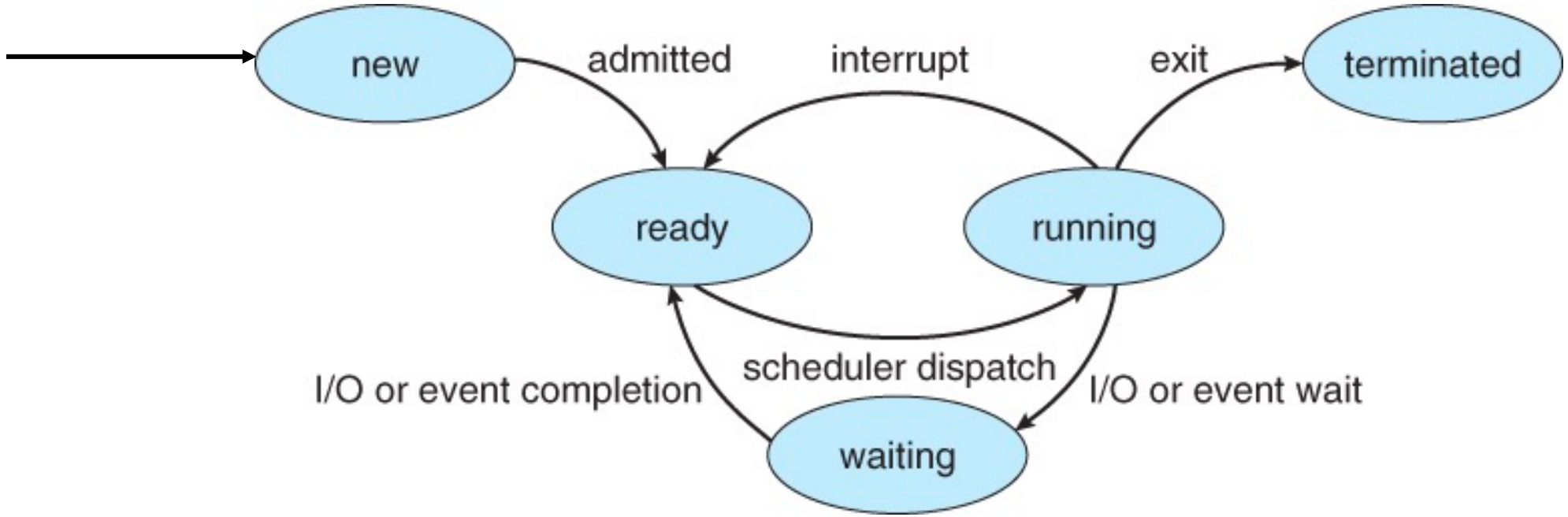
Tanu Malik

School of Computing, DePaul University

Visiting Faculty, CSE, IIT, Delhi



# Kernel Process States




- Involuntary termination from ready and wait state.

# Parent-child state

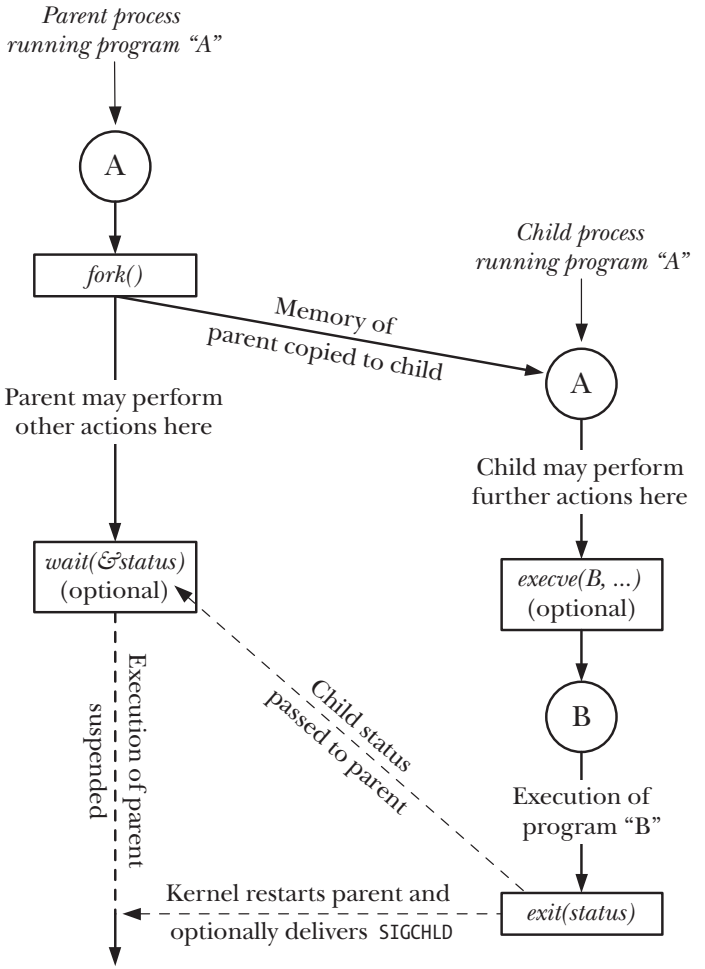



**Figure 24-1:** Overview of the use of *fork()*, *exit()*, *wait()*, and *execve()*

# Process Creation/ Coordination

- fork()
  - Create a child process
  - Identical to parent EXCEPT for return value of fork() call
  - Determines child/parent
- getpid() / getppid()
  - Get process ID of the currently running process
  - Get parent process ID
- exec() family
  - Replace currently running process with a different image
  - Process becomes something else losing previous code
  - Focus on execvp()
- wait() / waitpid()
  - Wait for any child to finish (wait)
  - Wait for a specific child to finish (waitpid)
  - Get return status of child

# Waiting for a child to finish – wait()

**#include <sys/types.h>**

**#include <wait.h>**

**pid_t wait(int *status);**

- Suspends/blocks calling process until child has finished
- Allow parent to be able to monitor the children to find out when and how they terminate.
- Returns:
  - Process ID of **a** terminated child on success
  - -1 on error, sets **errno**
- Parameters:
  - **status**: is a memory buffer set by **wait** in which termination status of child is populated, and evaluated using specific macros defined for **wait**.

# Example n.c

- Observe wait for each child by a parent
- child_wait.c
- child_status.c
- child_allstatus.c

# Wait() limitations

- The *wait()* system call has a number of limitations:
  - If a parent process has created multiple children, it is not possible to *wait()* for the completion of a specific child; we can only wait for the next child that terminates.
  - If no child has yet terminated, *wait()* always blocks. Sometimes, it would be preferable to perform a nonblocking wait so that if no child has yet terminated, we obtain an immediate indication of this fact.

# Waiting for specific child to finish—waitpid()

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t waitpid(pid_t pid, int *statloc, int options)
```

- Returns:
  - process ID : if OK,
  - 0 : if non-blocking option && no zombies around
  - -1 : on error
- Parameters:
  - Pid o child process
  - statloc: status
  - options

# wait() Vs waitPID()

| Wait() | Waitpid() |
| --- | --- |
| wait blocks the caller until **a** child process terminates | waitpid can be either blocking or non-blocking:If *options* is 0, then it is blocking<br>If *options* is WNOHANG, then is it non-blocking |
| if more than one child is running then wait() returns the first time **one of the parent's offspring exits** | waitpid is more flexible:<br>If *pid == -1*, it waits for any child process. In this respect, waitpid is equivalent to wait<br>If *pid > 0*, it waits for the child whose process ID equals pid<br>If *pid == 0*, it waits for any child whose process group ID equals that of the calling process<br>If *pid < -1*, it waits for any child whose process group ID equals that absolute value of pid |

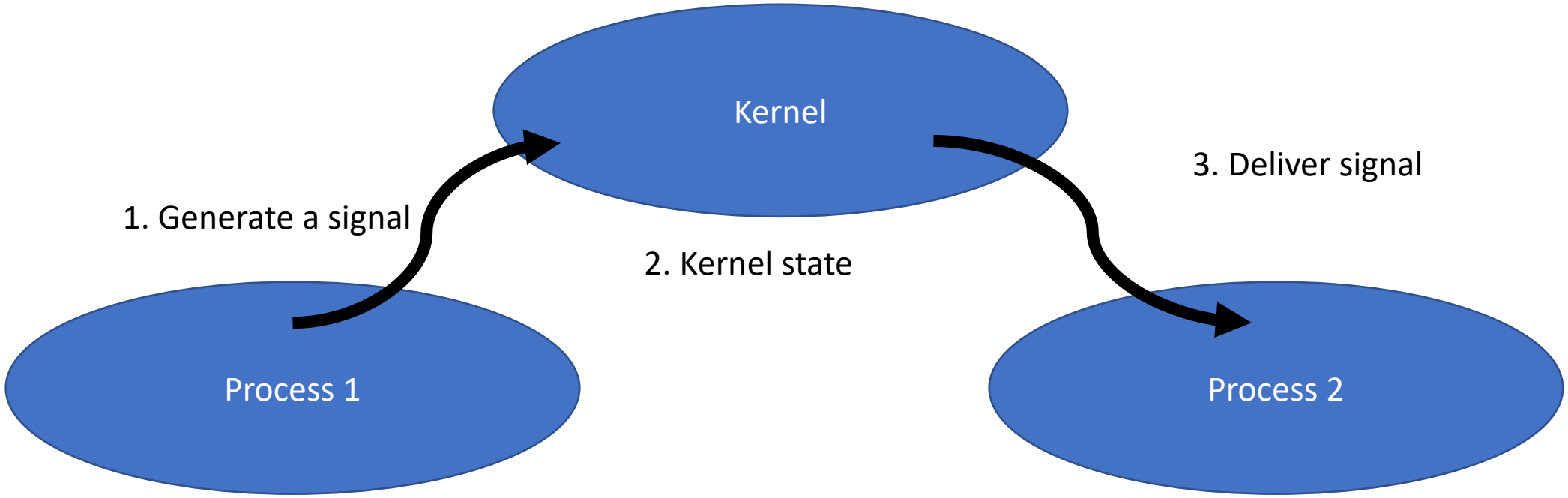
# Example

- Observe waiting for a specific child
- Get status of the exited child
- Several children

# Definition of Signal

- Signal: A notification of an event
  - Event gains attention of the OS
  - OS stops the application process immediately, sending it a signal
  - Default action for that signal executes
    - Can install a signal handler to change action
  - Application process resumes where it left off

# Flow of actions




A pending signal is delivered to a process (here Process 2) as soon as it is next scheduled to run,
or immediately if the process is already running

# Execution Flow with Signal Handler

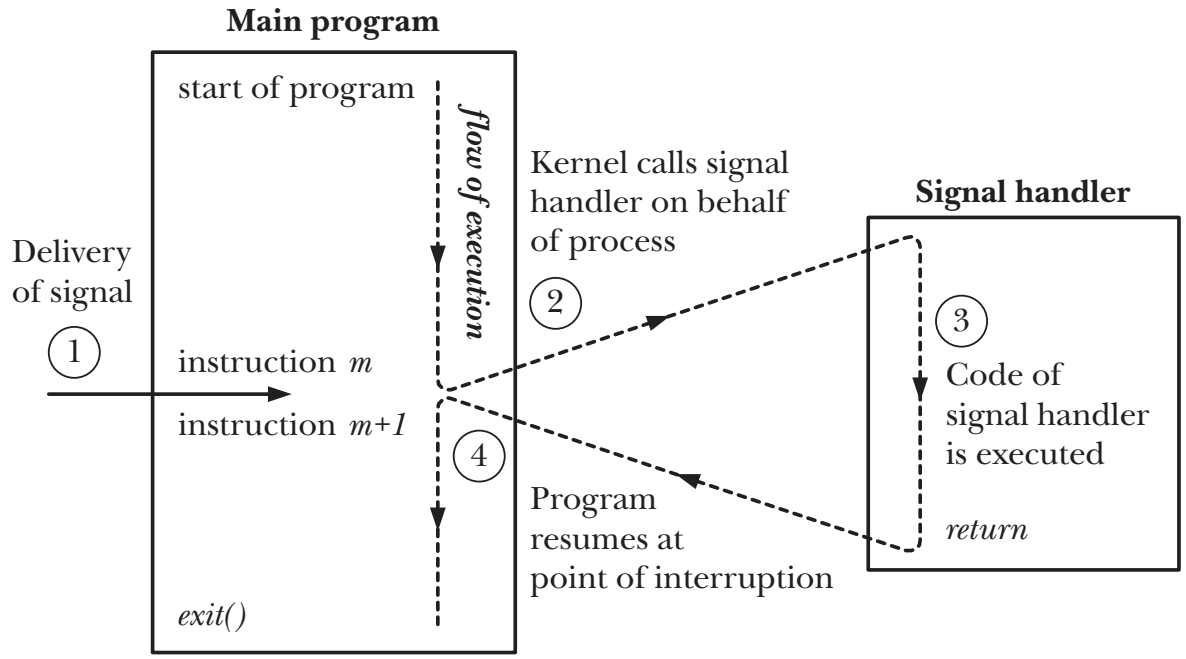



**Figure 20-1:** Signal delivery and handler execution

- Handler code is within the main program

# Some important signals (man 7 signal )

| Reason | Name | Default Action | Usual Use |
|---|---|---|---|
| User types Ctrl-C | SIGINT | Terminate (Can be caught) | Stop a process nicely |
| Kill -2 | SIGQUIT | Terminate (Can be caught) | Stop a process harshly producing a core dump (https://sigquit.wordpress.com/tag/core_pattern/) |
| Ctrl-Z | SIGSTOP | Stop Process (Cannot be caught) | Suspends a process |
| fg | SIGCONT | Continues a process | Starts after a stop |
| Kill -9 | SIGKILL | Terminate Process (Cannot be caught) | You want the process gone with no memory image |
| Process makes illegal memory reference | SIGSEGV | Terminate | Generated by buggy program |
| Division by 0 | SIGFPE | Floating point exception | Unsupported operation |

# Signal Flow

- Sending Signals
- Handling Signals
- Blocking Signals

# Sending Signals

1. Via Keystrokes
   - Ctrl-c -> 2/SIGINT signal
     - Default action is “terminate”
   - Ctrl-z -> 20/SIGTSTP signal
     - Default action is “stop until next 18/SIGCONT”
   - Ctrl-\ -> 3/SIGQUIT signal –Default action is “terminate”
2. Via Commands
   - kill –2 1234 or kill -SIGINT 1234
     - Same as pressing Ctrl-c if process 1234 is running in foreground
3. Via System Calls
   - In your program through kill() or raise() function
   - Example: raise(SIGSTOP)

# Via System Calls

- raise()

int raise(int iSig);

  - Commands OS to send a signal of type iSig to the **current** process
  - Returns 0 to indicate success, non-0 to indicate failure

  Example

```
int ret = raise(SIGINT); /* Process commits suicide. */
assert(ret != 0);      /* Shouldn't get here. */
```

# Via System Calls

- **kill()**

**int kill(pid_t iPid, int iSig);**

- Sends a **iSig** signal to the process whose id is **iPid**
- Equivalent to **raise(iSig)** when **iPid** is the id of current process

- Example

pid_t iPid = getpid(); /* Process gets its id.*/

kill(iPid, SIGINT); /* Process sends itself a SIGINT signal (commits suicide) */

# Kill

- #include <signal.h>
  int **kill**(pid_t *pid*, int *sig*);
  - Returns 0 on success, or –1 on error
- If *pid* > 0, the signal is sent to the process with the process ID specified by *pid*.
- If *pid* = 0, the signal is sent to every process in the same process group as the calling process, including the calling process itself.
- If *pid* < –1, the signal is sent to all of the processes in the process group whose ID equals the absolute value of *pid*.
- If *pid* = –1, the signal is sent to every process for which the calling process has permission to send a signal, except *init* (process ID 1) and the calling process. If a privileged process makes this call, then all processes on the system will be signaled, except for these last two.

# Handling Signals

- Each signal type has a default action
  - Terminate
  - Ignore
  - Generate core dump
  - Stop execution
  - Resume execution
- A program can install a signal handler to change action of (almost) any signal type
- Signal handler should be designed to be as simple as possible.

# Uncatchable signals

Special cases: A program cannot install a signal handler for signals of type:

- 9/SIGKILL – Default action is “terminate”
  - Catchable termination signal is 15/SIGTERM
  - To kill ”wild” processes or perform full system shutdown process, which sends TERM first and if not then SIGKILL
- 19/SIGSTOP – Default action is “stop until next 18/SIGCONT”
  - Catchable suspension signal is 20/SIGTSTP

# Catchable signals

- `SEGV:` accessing an illegal memory address

- `BUS`: accessing a memory address with invalid alignment.

- Catchable signals
- On catching perform some cleanup - kill child processes, perhaps remove temporary files, etc.

# Installing a Signal Handler

- signal()

  sighandler_t signal(int iSig, sighandler_t pfHandler);

- Installs function pfHandler as the handler for signals of type iSig
- pfHandler is a function pointer: typedef void (*sighandler_t)(int)
- After call, (*pfHandler) is invoked whenever process receives a signal of type iSig

# Examples

- Catch-Ctrl-c.c: single handler for SIGINT

- Establishing the same handler for two different signals

# Predefined Signal Handler: SIG_IGN

- Predefined value: **SIG_IGN**
  Can use as argument to signal() to **ignore** signals

```
int main(void) {
void (*pfRet)(int);
pfRet = signal(SIGINT, SIG_IGN); ...
}
```

- Subsequently, process will ignore 2/SIGINT signals

# Predefined Signal Handler; SIG_DFL

- Predefined value: **SIG_DFL**

- Can use as argument to signal() to **restore default action**

```
int main(void) {
void (*pfRet)(int);
pfRet = signal(SIGINT, somehandler); ...
pfRet = signal(SIGINT, SIG_DFL);
...
}
```

- Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals (“terminate”)

# Example

- Signal-predefined.c

# Pending signals

- Why? There is a brief period of time between the time a signal is generated and the time a signal is delivered (i.e. the action for the signal is taken). If another signal in generated during this time problems can arise.
- Problematic situation as race conditions may arise.

# Blocking Signals

- To block a signal is to queue it for delivery at a later time
- Differs from ignoring a signal
- Each process has a signal mask in the kernel
- OS uses the mask to decide which signals to deliver
- User program can modify mask with sigprocmask()

# Blocking Signals in General

- sigprocmask()

int sigprocmask(int iHow, const sigset_t *psSet, sigset_t *psOldSet);

- psSet: Pointer to a signal set
- psOldSet: (Irrelevant for our purposes)
- iHow: How to modify the signal mask
  - SIG_BLOCK: Add psSet to the current mask
  - SIG_UNBLOCK: Remove psSet from the current mask
  - SIG_SETMASK: Install psSet as the signal mask
  - Returns 0 iff successful
- Functions for constructing signal sets
  - sigemptyset(), sigaddset(), …

# Example

```
sigset_t sSet;
int main(void) {
int iRet;
sigemptyset(&sSet);
sigaddset(&sSet, SIGINT);
iRet = sigprocmask(SIG_BLOCK, &sSet, NULL);
assert(iRet == 0);
if (iFlag == 0) {
/* Do something */
}
iRet = sigprocmask(SIG_UNBLOCK, &sSet, NULL);
assert(iRet == 0);
…
}
```

# Blocking Signals in Handlers

- How to block signals when handler is executing?
  - While executing a handler for a signal of type x, all signals of type x are blocked automatically
  - When/if signal handler returns, block is removed
- Additional signal types to be blocked can be defined at time of handler installation…

# Example

- count-ctrl-c.c

# Installing a Handler with Blocking

sigaction()

**int sigaction(int iSig, const struct sigaction *psAction, struct sigaction *psOldAction);**

- iSig: The type of signal to be affected
- psAction: Pointer to a structure containing instructions on how to handle signals of type iSig, including signal handler name and which signal types should be blocked
- psOldAction: (Irrelevant for our purposes)
- Installs an appropriate handler
- Automatically blocks signals of type iSig
- Returns 0 iff successful

Note: More powerful than signal()

# The special case of init

- The *init* process (process ID 1), which runs with user and group of *root*, is a special case. It can be sent only signals for which it has a handler installed. This prevents the system administrator from accidentally killing *init*, which is fundamental to the operation of the system.

# Isolation

# What is a jail?

Is the prisoner isolated?

# Isolation

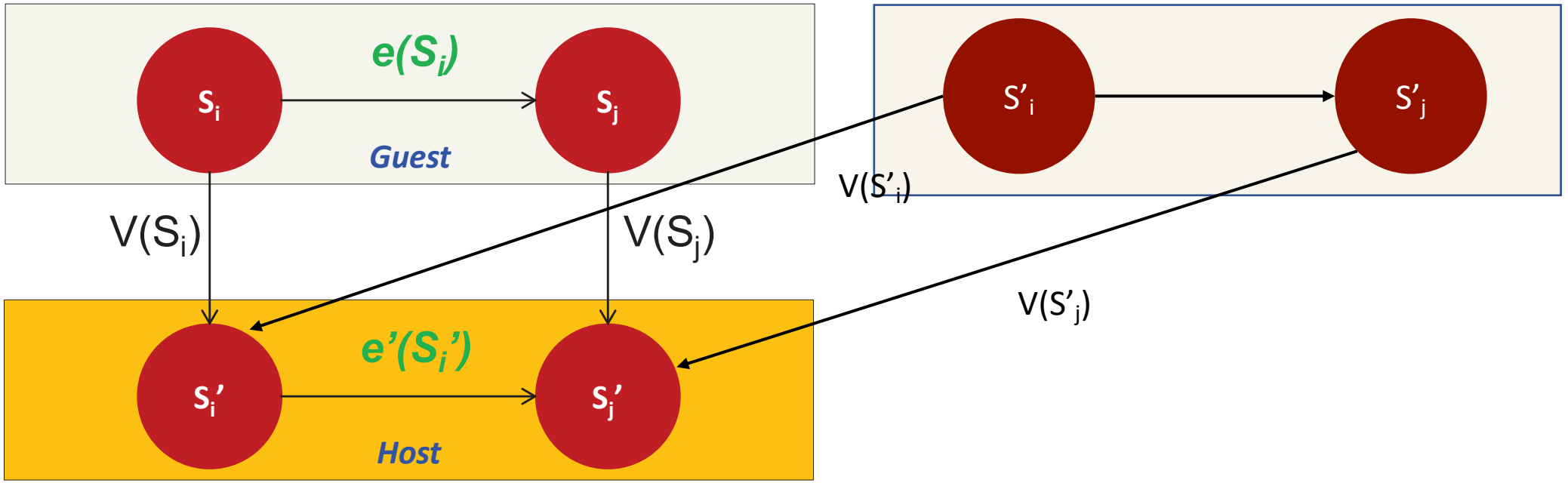
- Property 1: Isolation is as strong as the mechanism of isolation.
- Property 2: Even a strong isolation may also get compromised.

# Isolation

- Isolation in a computing system is a mechanism or protocol to restrict an object A from influencing operation of object B and vice versa.

# Isolation

- Will another process's actions cause the host OS to make the exactly same transition that is supposed to be made by the guestOS?
- What does hostOS need to verify the transition?



# Process Isolation

- There are two primary methods:
- Credentials and Privileges: By determining the identity of the user executing the process and their privilege?
- Access control: And what permissions do they have?

# Access Control via Permissions

- UNIX is a multi-user system.
  - Every file and directory in your account can be protected from or made accessible to other users by changing its access permissions.
- Every user has responsibility for controlling access to their files.

# Permissions

- Permissions for a file or directory may be any or all of:

**r - read (4)**

**w - write (2)**

**x – execute (1) = running a program**

- Each permission (rwx) can be controlled at three levels:

**u - user = yourself**

**g - group = can be people in the same project**

**o - other = everyone on the system**

# Process Credentials

- Numerical identity of the process within userspace.
- Credentials can be changed by acquiring and loosing privileges.

# Determining Identity

- whoami: name associated with current uid
  - Each user has a unique UID
  - root is designated superuser with uid = 0

- Id is associated with a credential.

# Determining Process Credentials

- Every process has a set of associated numeric user identifiers (UIDs) and group identifiers (GIDs).

- These identifiers are as follows:
  - real user ID and group ID;
  - effective user ID and group ID;
  - saved set-user-ID and saved set-group-ID;

- 3 user IDs: ruid, euid, suid
- 3  groupids: rgid, egid, sgid

# Real User ID and Read Group ID

- The real user ID and group ID identify the user and group to which the process belongs.
  - A login shell gets its real user and group IDs from the third and fourth fields of the user's password record in the /etc/passwd file
- When a new process is created (e.g., when the shell executes a program), it inherits these identifiers from its parent.

# Effective User ID and Effective Group ID

- Used to determine the privilege granted to a process when it tries to perform various operations
  - Privilege granted to a process when it accesses resources such as files and System V interprocess communication (IPC) objects, which themselves have associated user and group IDs determining to whom they belong.
  - The effective user ID is also used by the kernel to determine whether one process can send a signal to another.
- Process with effective userid = 0 has all the privileges of a superuser.
  - Certain system calls can be executed only by privileged processes.

# Changing EID and EGID

- Normally, the effective user and group IDs have the same values as the corresponding real IDs.
- Two ways in which the effective IDs can assume different values.
  - use of system calls
  - execution of set-user-ID and set-group-ID programs

# Effective User ID and Effective Group ID

- When a process executes a file by execve it keeps its 3 userids unless the set user id bit on the file is set in which case:
  - If the file executed is a set-userID file, the effective and saved user IDs of the process are set to the owner of the file executed.
  - If the file executed is a set-group-ID file, the effective and saved group IDs of the process are set to the group of the file executed.
- If the file executed is not a set-user-ID or set-group-ID file, the effective user ID, saved user ID, effective group ID, and saved group ID are not changed.

# Saved set user ID and Saved set group-ID

If the set-user-ID (set-group-ID) permission bit is enabled on the executable, then

- the effective user (group) ID of the process is made the same as the owner of the executable.

If the set-user-ID (set-group-ID) bit is not set, then

- no change is made to the effective user (group) ID of the process.
- The values for the saved set-user-ID and saved set-group-ID are copied from the corresponding effective IDs. This copying occurs regardless of whether the set-user-ID or set-group-ID bit is set on the file being executed.