Process Creation

References

1. `Computer Systems A Programmer's Perspective', Randal Bryant and David O'Hallaron, Pearson Education
2. Unix System Programming, Keith Haviland, Dina Gray and Ben Salama, Addison-Wesley

Introduction

- Unix Provides a library routine called system which allows a shell command to be executed from within a program.
- One can invoke a standard shell as an intermediary, rather than attempt to run the command directly
- \- c argument used in the invocation of the shell tells it to take commands from the next string argument, rather than standard input.

Introduction

- Define PATH properly.
- Execute any shell script, you have developed as under:

```
bash-2.03$ cc -o execcommand execcommand.c
bash-2.03$ execcommand

Enter command to be executed: filecopy
Enter the filename(s) to be copied
execcommand.c
Enter the destination for file(s) to be copied :
execcommand.bak
command execution completed
```

$ ls –la

Shell creates a process to run this command

$ ls –la | more

- Two processes will be created to run this command
- Several processes can run concurrently run the same program

Control Flow

- Computers do Only One Thing
  - From startup to shutdown, a CPU simply reads and executes (interprets) a sequence of instructions, one at a time.
  - This sequence is the system's physical *control flow* (or *flow of control*).

Physical control flow

Time

<startup>
inst_1
inst_2
inst_3
…
inst_n
<shutdown>

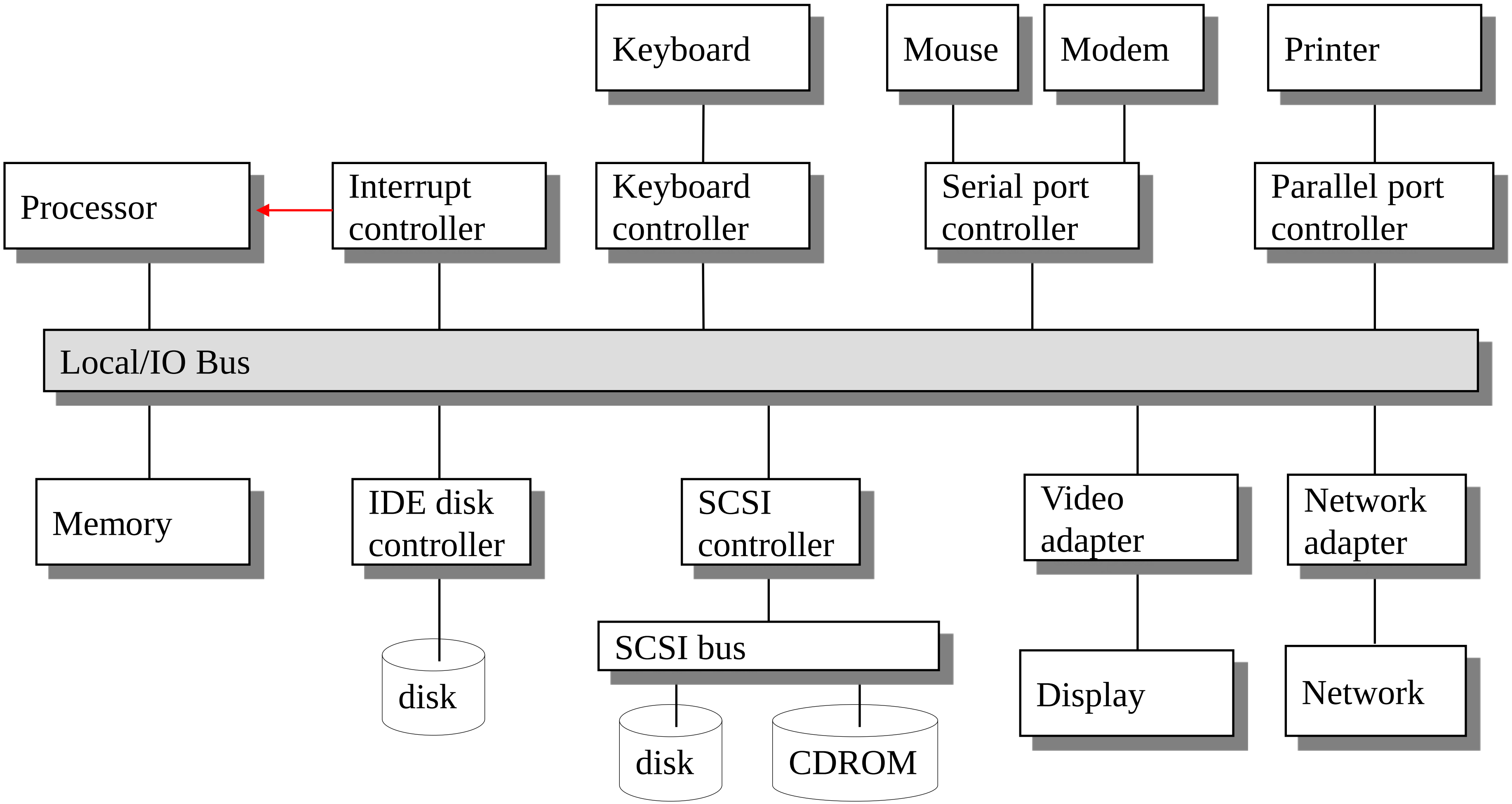
Altering the Control Flow

- Up to Now: two mechanisms for changing control flow:
  - Jumps and branches
  - Call and return using the stack discipline.
  - Both react to changes in program state.
- Insufficient for a useful system
  - Difficult for the CPU to react to changes in system state.
    - data arrives from a disk or a network adapter.
    - Instruction divides by zero
    - User hits ctl-c at the keyboard
    - System timer expires
- System needs mechanisms for "exceptional control flow"

Exceptional Control Flow

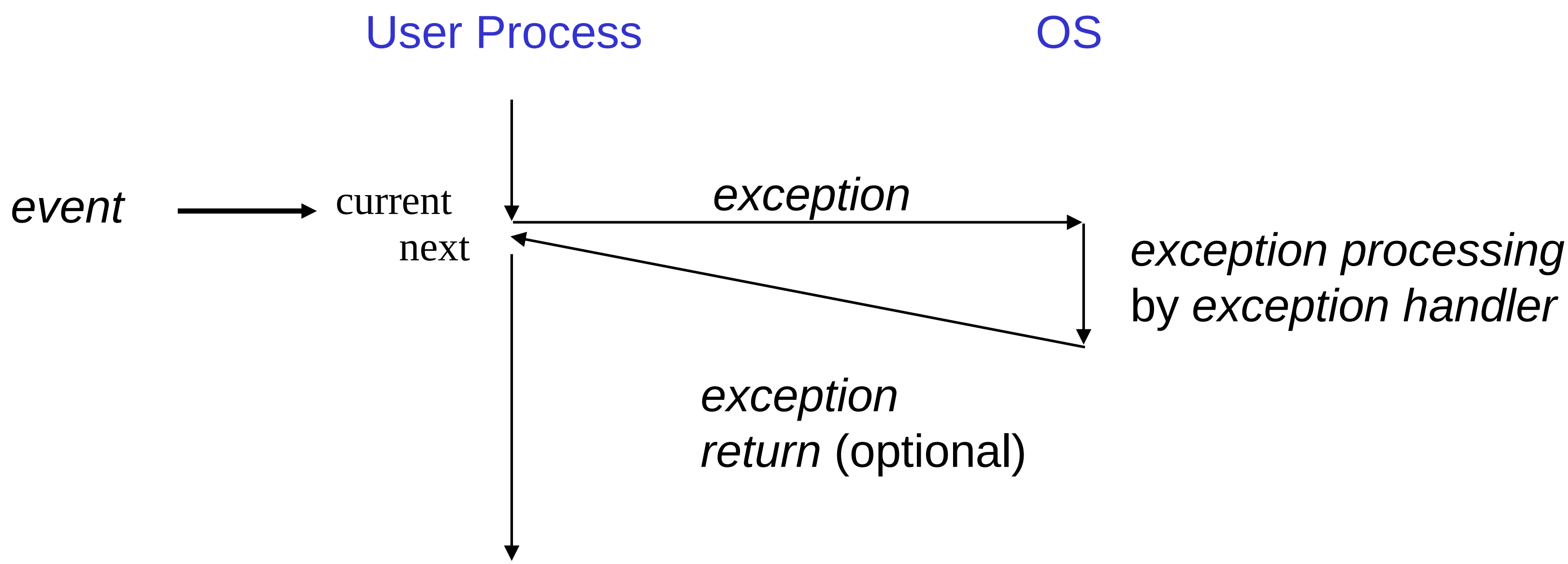
  - Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level Mechanism
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context switch
  - Signals
  - Nonlocal jumps (setjmp/longjmp)
  - Implemented by either:
    - OS software (context switch and signals).
    - C language runtime library: nonlocal jumps.

System context for exceptions

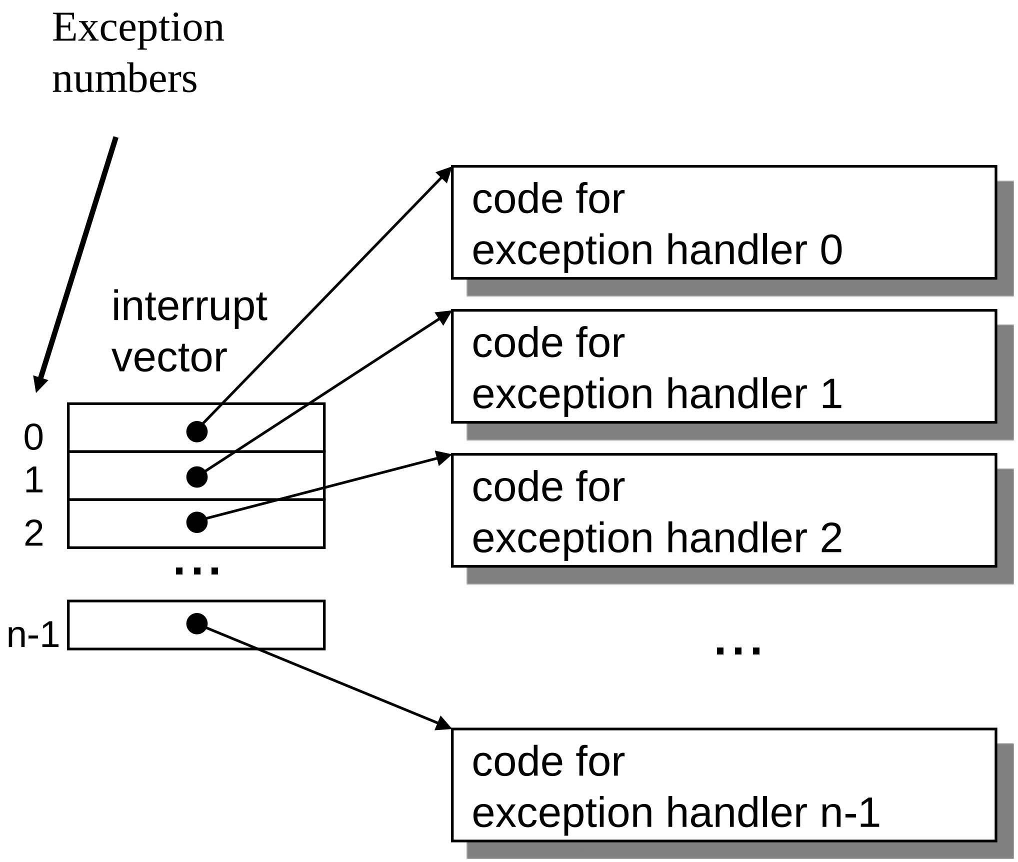


Exceptions

- An *exception* is a transfer of control to the OS in response to some *event* (i.e., change in processor state)



Interrupt Vectors



- Each type of event has a unique exception number k
- Index into jump table (a.k.a., interrupt vector)
- Jump table entry k points to a function (exception handler).
- Handler k is called each time exception k occurs.

Asynchronous Exceptions (Interrupts)

- Caused by events external to the processor
  - Indicated by setting the processor's interrupt pin
  - handler returns to "next" instruction.
- Examples:
  - I/O interrupts
    - hitting ctl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button
  - Soft reset interrupt
    - hitting ctl-alt-delete on a PC

Synchronous Exceptions

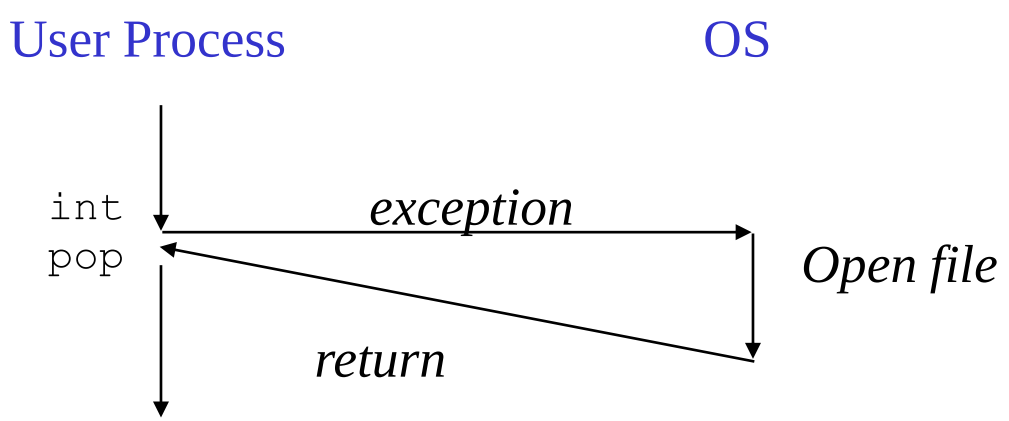
- Caused by events that occur as a result of executing an instruction:
  - Traps
    - Intentional
    - Examples: system calls, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - Faults
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable).
    - Either re-executes faulting ("current") instruction or aborts.
  - Aborts
    - unintentional and unrecoverable
    - Examples: parity error, machine check.
    - Aborts current program

Trap Example

- Opening a File
  - User calls `open(filename, options)`

```
0804d070 <__libc_open>:
 . . .
 804d082:       cd 80                     int    $0x80
 804d084:       5b                        pop    %ebx
 . . .
```

- Function open executes system call instruction int
  - OS must find or create file, get it ready for reading or writing
  - Returns integer file descriptor



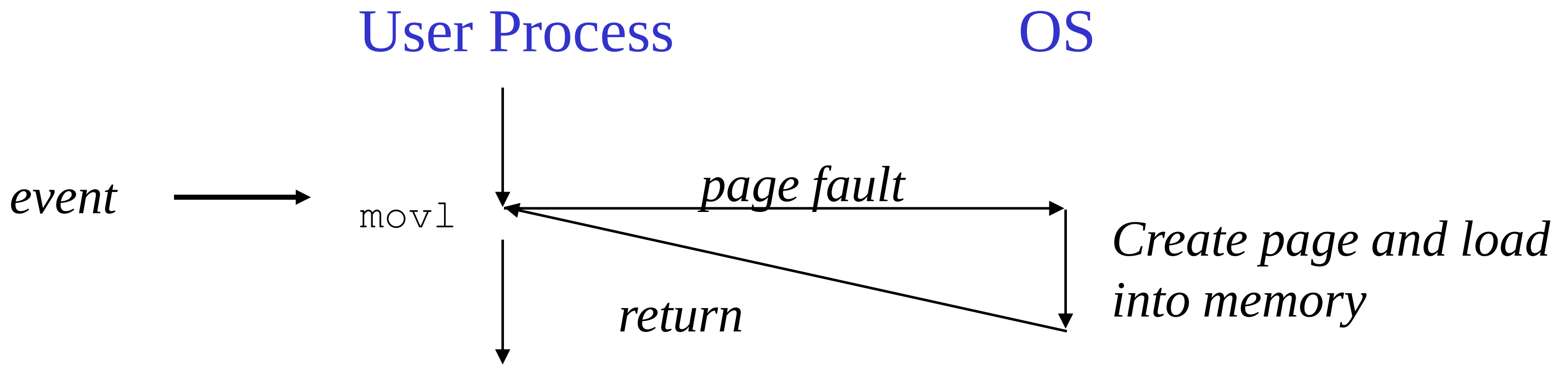
Fault Example #1

- Memory Reference
  - User writes to memory location
  - That portion (page) of user's memory is currently on disk

```
int a[1000];
main ()
{
    a[500] = 13;
}
```

```
80483b7:      c7 05 10 9d 04 08 0d   movl   $0xd,0x8049d10
```

  - Page handler must load page into physical memory
  - Returns to faulting instruction
  - Successful on second try



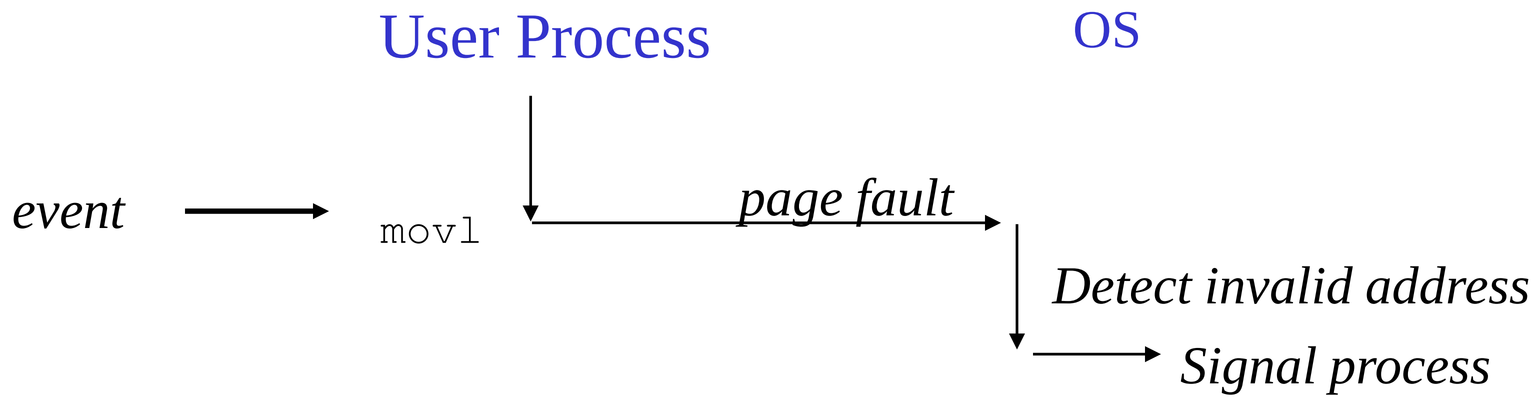
Fault Example #2

```
int a[1000];
main ()
{
    a[5000] = 13;
}
```

- Memory Reference
  - User writes to memory location
  - Address is not valid

```
80483b7:      c7 05 60 e3 04 08 0d   movl   $0xd,0x804e360
```

  - Page handler detects invalid address
  - Sends SIGSEG signal to user process
  - User process exits with “segmentation fault”



System Calls for
Process Creation and Manipulation

- fork: to create a new process by duplicating the calling process. The basic process creation primitive
- exec: A family of library routines and one system call, each of which performs the same underlying function: the transformation of a process by overlaying its memory space with a new program

System Calls for Process Creation and Manipulation

- wait: provides rudimentary process synchronization. It allows one process to wait until another related process finishes.
- exit: to terminate a process

fork system call

- Mechanism to transform unix into a multitasking system
- It causes the kernel to create new process called a “child process”
- A child process created with fork is an almost perfect copy of its parent.
- The child process will retain the values they held in the parent, except the return value from fork itself.
- Data available to the child occupies a different absolute place in memory, thus subsequent changes in one process will not affect the variables in the other.

fork system call

- After fork, the parent process and child process will to continue to execute in parallel/ concurrently.
- Both will resume execution at the statement immediately after the call to fork.

```
#include <sys/types.h>
#include <unistd.h> /* for fork() */
main()
{
pid_t pid;      /*holds process-id in parent*/
printf("One\n");
pid=fork();
printf("Two\n");
}
[sanjay@dslabsrv17 unixprgs]$ fork01
One
Two
Two
```

- Before fork, process A is existing
- After fork process A and Process B will exist, B is the new process spawned by the call to fork and two processes will exist

fork system call

- `fork` is called without arguments and returns `pid_t`, which is an integer
- The value of pid distinguishes parent and child.
- In parent, pid is set to non-zero, positive number.
- In child it is set to zero.
- Return value in parent and child differs, the programmer is able to specify different actions for the processes.
- The number returned to the parent in pid is called process-id of the child

fork system call

- Both the processes will run concurrently and without synchronization.
- Fork is useful when parent and child perform different but related tasks, cooperating by using one of the unix inter-process communication mechanisms such as signals or pipes.

```
[sanjay@dslabsrv17 unixprgs]$ cc -o runforkexecl
  runforkexecl.c
[sanjay@dslabsrv17 unixprgs]$ runforkexecl
```

Inherited data and file descriptors

- All the files open in the parent process are also open in the child process.
- The child will maintain its own copy of the file descriptors associated with each file.
- However, files kept open across a call to fork remain intimately or closely connected in child and parent.
- This is because the read-write pointer for each file is maintained by the system, it is not embedded explicitly within the process itself.

Inherited data and file descriptors

- When a child process moves forward in a file, the parent process will also find itself at the new position

Question:

- What will happen within a parent process when a child process closes a file descriptor inherited across a fork?

exec and open files

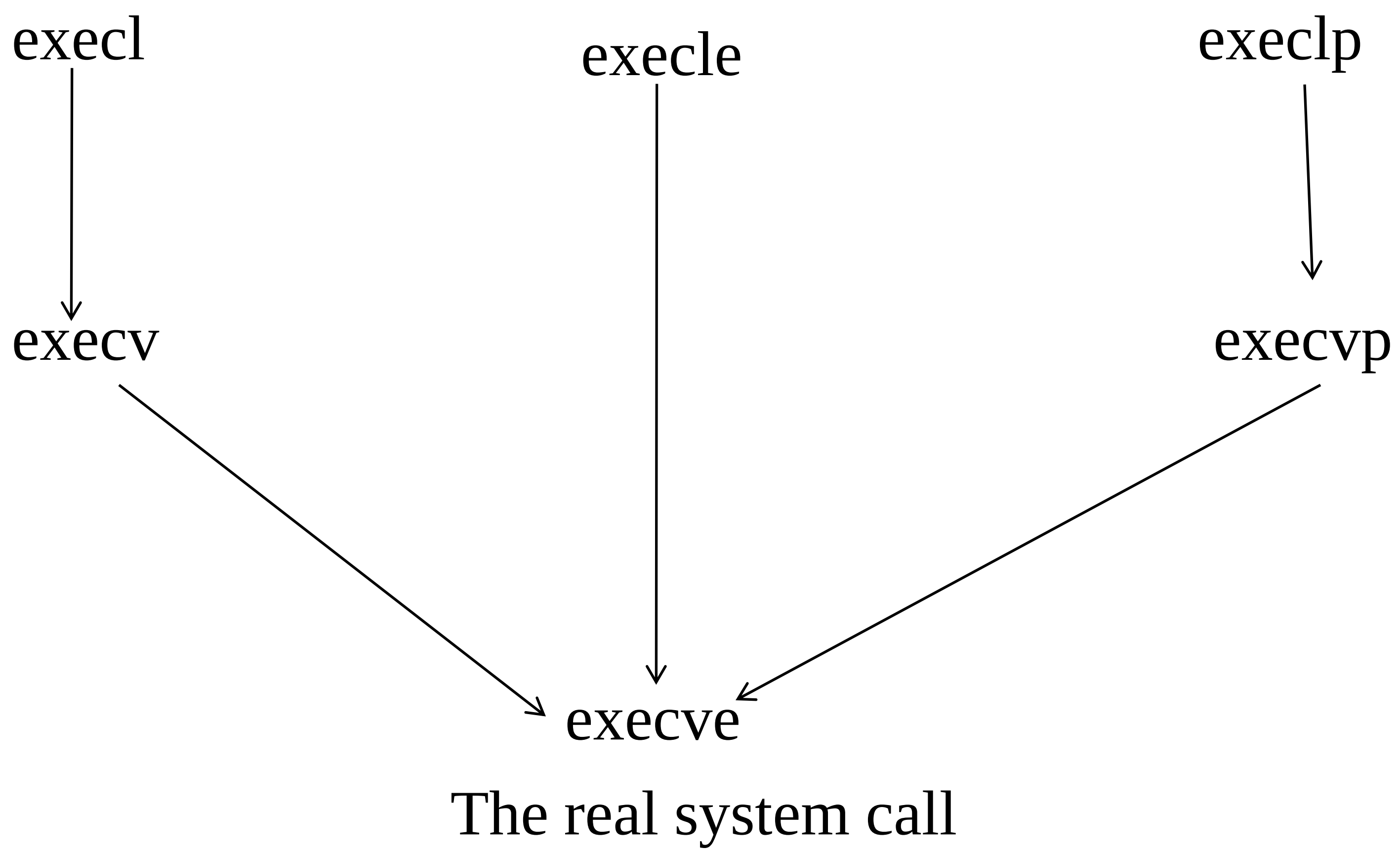
- Open file descriptors are also normally passed across calls to exec.
- Files open in the original program are kept open when an entirely new program is started through exec.
- The read-write pointers for such files are unchanged by the exec call.
- **`fcntl`** function can be used to set the close-on-exec flag associated with a file.

Running new program with exec

- To initiate the execution of a new program
- All varieties of exec performs the same function:
  - Transform the calling process by loading a new program into its memory space
- If exec is successful, the calling program is completely overlaid by the new program, executed from its beginning
- The new process retains process-id of the calling process
- Exec does not create a new subprocess to run concurrently with the calling program
- There is no return from a successful call to exec

exec family tree

The difference is in the way parameters can be passed to these functions



exec system call

- There are various forms of the exec system call:
- The execl forms use a null-terminated list of arguments;
  - execl() needs a full pathname for the progran to execute
  - execlp() makes use of the user's search path so only needs a filename of the executable
  - The execv() forms need an array of arguments rather than a list;

exec system call

- As with the execl, there is an execp() from that uses the search path.
- exec calls overwrite the existing process so there is no return value from a successful call.
- The arguments to exec form the arguments to main() of the executed program.

Use of exec and fork

- By forking and then using exec within the newly created child, a program can run another program within a subprocess and without obliterating or destroying itself
- Refer following programs:
  - exec01.c
  - execlfail.c
  - execlfail.c

```
#include <sys/types.h>
#include <sys/wait.h> /* for wait() */
#include <unistd.h> /* for fork() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for perror() */

int main(void)
{
int fatal(char *);
pid_t pid;
```

```
switch(pid = fork())
{ case -1:
        fatal("fork failed");
        break;
case 0:
        /* child process calls exec */
        execl("/bin/ls", "ls", "-l", (char *)0);
        fatal("exec failed");
        break;
default:
        /* parent process uses wait to suspend execution
        * until child process finishes */
        wait((int *)0);
        printf("ls completed\n");
        exit(0);
}
}
int fatal(char *s)
{       perror(s);
        exit(1);        }
```

Terminating processes with the exit system call

- Exit system call is used to terminate a process.
- A process will also stop when it runs out of program by reaching the end of the main function, or when main executes a return statement.
- The single, integer argument to exit is called the process' exit status, the low-order eight bits of which are available to the parent process, providing it has executed a wait system call.
- The value returned through exit in this way is normally used to indicate the success or failure of the task performed by the process.

Terminating processes with the exit system call

- By convention a process returns zero on normal termination, some non-zero value if something has gone wrong.

`exit()` has a number of other consequences:

  - All open file descriptors are closed.
  - If the parent process has executed a wait call, it will be restarted.
  - Exit will also call any programmer-defined exit handling routines and perform what are generally described as clean-up actions, e.g. be concerned with buffering in the standard I/O library.
  - A programmer can also set at least 32 bit handling routines with the `atexit` function.

Synchronizing processes

The `wait()` system call

- wait temporarily suspends the execution of a process while a child process is running.
- Once the child process is finished, the waiting parent is restarted.
- If more than one child is running then wait returns as soon as any one of the parent's offspring exits.
- wait is often used by a parent process just after a call to fork, e.g.
- Refer: `runforkexecl.c`

Synchronizing processes

- The combination of fork and wait is most useful when the child process is intended to run a completely different program by calling exec.
- The return value from wait is normally the process-id of the exiting child.
- If wait returns (pid_t)-1, it can mean that no child exists and in this case errno will contain the error code ECHILD.
- The parent can sit in a loop waiting for each of its offspring.
- When the parent realizes that all the children have terminated, it can continue.

```c
#include <sys/types.h>
#include <sys/wait.h> /* for wait() */
#include <unistd.h> /* for fork() */
#include <stdio.h> /* for printf() */
#include <stdlib.h> /* for perror() */

int main(void)
{
int fatal(char *);
pid_t pid;

printf("PID before fork, ie PID of current
process: %d\n", getpid());
switch(pid = fork())
{
case -1:
        fatal("fork failed");
        break;
```

Parent and Child Process IDs known to each other

```
case 0:
        /* child process calls exec */
        printf("Message from a Child Process\n");
        printf("\n");
        printf("The Value PID assigned by pid-t and
known to a Child  Process : %d\n", pid);
        printf("The value of PID of child process:
%d\n", getpid());
        printf("The value of Parent PID, who created me:
%d\n", getppid());

        execl("/bin/ls", "ls", "-l", (char *)0);
        printf("\n");

        fatal("exec failed");
        break;
```

```
default:
        /* parent process uses wait to suspend execution
        * until child process finishes */
        wait((int *)0);
        printf("Message from a parent process\n");
        printf("PID of child known to the parent: %d\n", pid);
        printf("TASK ACCOMPLISHED, ie. We are back from MOON
\n");
        exit(0);
}
```

[sanjay@dslabsrv17 unixprgs]$ cc -o runforkexecltest runforkexecltest.c

[sanjay@dslabsrv17 unixprgs]$ runforkexecltest

PID before fork, ie PID of current process: **24097**

Message from a Child Process

The Value PID assigned by pid-t and known to a Child Process : **0**

The value of PID of child process: **24098**

The value of Parent PID, who created me: **24097**

Output of ls command will be displayed here

Message from a parent process

PID of child known to the parent: **24098**

TASK ACCOMPLISHED, ie. We are back from MOON

Synchronizing processes

- wait takes one argument status, a pointer to an integer.
- If the pointer is NULL then the argument is simply ignored.
- If wait is passed a valid pointer, status will contain useful status information when wait returns. This information will be the exit-status of the child passed through exit.
- Refer: `status01.c`

Synchronizing processes

- The value returned to the parent via exit is stored in the high-order eight bits of the integer status.
- To be meaningful the low-order bits must be zero
- WIFEXITED macro defined in <sys/wait.h> tests to see if this is in fact the case. It returns the value stored in the high-order bits of status
- If it returns 0 then the child process was stopped in its track by another process using IPC mechanism called **signal**

Waiting for a particular child process: waitpid

- The exit system call is used to terminate a process.
- A process also stop when it runs out of program by reaching the end of the main function, or when main executes a return statement.
- The single, integer argument to exit is called the process' exit status, the low-order eight bits of which are available to the parent process, providing it has executed a wait system call.

Waiting for a particular child process: waitpid

- The value returned through exit in this way is normally used to indicate the success or failure of the task performed by the process.
- By convention, a process returns zero on normal termination.
- Some non-zero value indicates something has gone wrong.
- exit will close all open file descriptors, if the parent process has executed a wait call, it will be restarted.

Waiting for a particular child process: waitpid

- A process waits for its children to terminate or stop by calling the waitpid function.

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

- First argument, pid specifies the process-id of the child process that the parent wishes to wait for.
- If it is set to -1 and the options argument is set to 0, then waitpid behaves exactly the way wait behaves.
- -1 indicates an interest in any child process

Waiting for a particular child process: waitpid

- If pid is greater than 0 then the parent waits for the child with a process-id of pid.
- Status will hold the status of the child process when waitpid returns.
- The final argument, options, can take a variety of values defined in <sys/wait.h>. Options can take a variety of values defined in <sys/wait.h>. WNOCHANG is the most useful
- It allows waitpid to sit in a loop monitoring a situation but not blocking if the child process is still running.
- If WNOCHANG is set then waitpid will return 0 if the child has not yet terminated

Waiting for a particular child process: waitpid

- Refer: waitpid1.c, status03.c

Waiting for a particular child process: waitpid

- The default behaviour can be modified by setting options to various combinations of the WNOHANG and WUNTRACED.
- WNOHANG: return immediately (with a return value of 0) if none of the child processes in the wait set has terminated yet.
- WUNTRACED: Suspend execution of the calling process until a process in the wait set becomes terminated or stopped. Returns the PID of the terminated or stopped child that caused the return.

Waiting for a particular child process: waitpid

- WNOHANG | WUNTRACED: Suspend execution of the calling process until a child in the wait set terminates or stops, and then return the PID of the stopped or terminated child that caused the return. Also, return immediately (with a return value of 0) if none of the processes in the wait set is terminated or stopped.

Checking the Exit Status of a Reaped Child

- If the status argument is non-null, then waitpid encodes status information about the child that caused the return in the status argument. The wait.h include file defines several macros for interpreting the status argument:
- WIFEXITED(status): Returns true if the child terminated normally, via a call to exit or a return.
- WEXITSTATUS(status): Returns exist status of a normally terminated child. This status is only defined if WIFEXITED returned true.

Checking the Exit Status of a Reaped Child

- WIFSIGNALED(status): Returns true if the child process terminated because of a signal that was not caught.
- WTERMSIG(status): Returns the number of signal that caused the child process to terminate. This status is only defined if WIFSIGNALED(status) returned true.
- WIFSTOPPED(status): Returns true if the child that caused the return is currently stopped.

Checking the Exit Status of a Reaped Child

- WSTOPSIG(status): Returns the number of the signal that caused the child to stop. This status is only defined if WIFSTOPPED(status) returns true.
- Error Conditions:
- If the calling process has no children, then waitpid returns -1 and sets errno to ECHILD. If the waitpid function was interrupted by a signal, then it returns -1 and set errno to EINTR.

Zombie processes and Premature exits

- A child exits when its parent is not currently executing wait
- A parent exits when one or more children are still running
- A zombie process occupies a slot in a table maintained by the kernel for process control, but does not use any other kernel resources

Process Attributes

- Process-id

```
pid=getpid();
ppid-getppid();
```

Process groups and process group-ids

- Unix allows processes to be placed into groups

```
who | awk '{print $1}' | sort -u
```

- It is useful when a set of processes are doing inter-process communication (IPC) with signals
- Each process group is denoted by a process group-id of type pid_t, i.e. pid_t getpgrp(void);
- Use getprgrp() system call to obtain process group-id.

Changing process group

- A process can be placed in a new process group

```
int setpgid(pid_t pid, pid_t pgid);
```

Sessions and session Ids

- Each process belongs to a session
- A session is a collection of a single foreground process group using the terminal and one or more background process groups

```
pid_t getsid(pid_t pid);
pid_t setsid(void);
```

- If is passed a value of 0 then it returns the session-id of the calling process otherwise session-id of the process identified by pid is returned.
- Useful for daemons, as they do not have controlling terminal. It can start a sessions and move into a new session.

The environment

- One can get the environment of a process by adding an extra parameter envp to the parameter list of the main function within a program
- Example: showenv.c
- The default environment is passed to a process that was created through a call to exec or fork.

Tools to manipulate processes

- Unix system provides a number of useful tools for monitoring and maipulating processes:
- strace: prints a trace of each system call invoked by a program and its children. You need to compile your program with –static to get a cleaner trace without a lot of output related to shared library
- ps: Lists processes (including zombies) currently in the system
- top: Prints information about the resource usage of current processes
- kill: Sends a signal to a process. Useful for debugging programs with signal handlers
- /proc: A virtual file system that exports contents of numerous kernel data structures in an ASCII form that can be read by user programs.

Tools to manipulate processes (cont)

```
[sanjay@dslab66 tmp]$ cc -static -o sigtalk sigtalk.c
[sanjay@dslab66 tmp]$ strace -p 28259
write(1, "\nPARENT: sending SIGQUIT\n\n", 26) = 26
rt_sigprocmask(SIG_BLOCK, [CHLD], [], 8) = 0
rt_sigaction(SIGCHLD, NULL, {SIG_DFL}, 8) = 0
rt_sigprocmask(SIG_SETMASK, [], NULL, 8) = 0
nanosleep({10, 0}, {10, 0})             = 0
munmap(0x40000000, 4096)                = 0
exit_group(0)                           = ?
```

To see current load average on your linux system

```
[sanjay@dslab66 tmp]$ cat /proc/loadavg
1.56 0.58 0.64 3/58 28264
```