Operating System

Shell, System Calls and Processes

Reference : [Stallings] Operating Systems- Internals and Design Principals, Fifth Edition, William Stallings

Shell

- A command Interpreter
- Different implementations
- The first Unix shell, Ken Thompson's sh
- Many others are developed on top of basic shell : sh

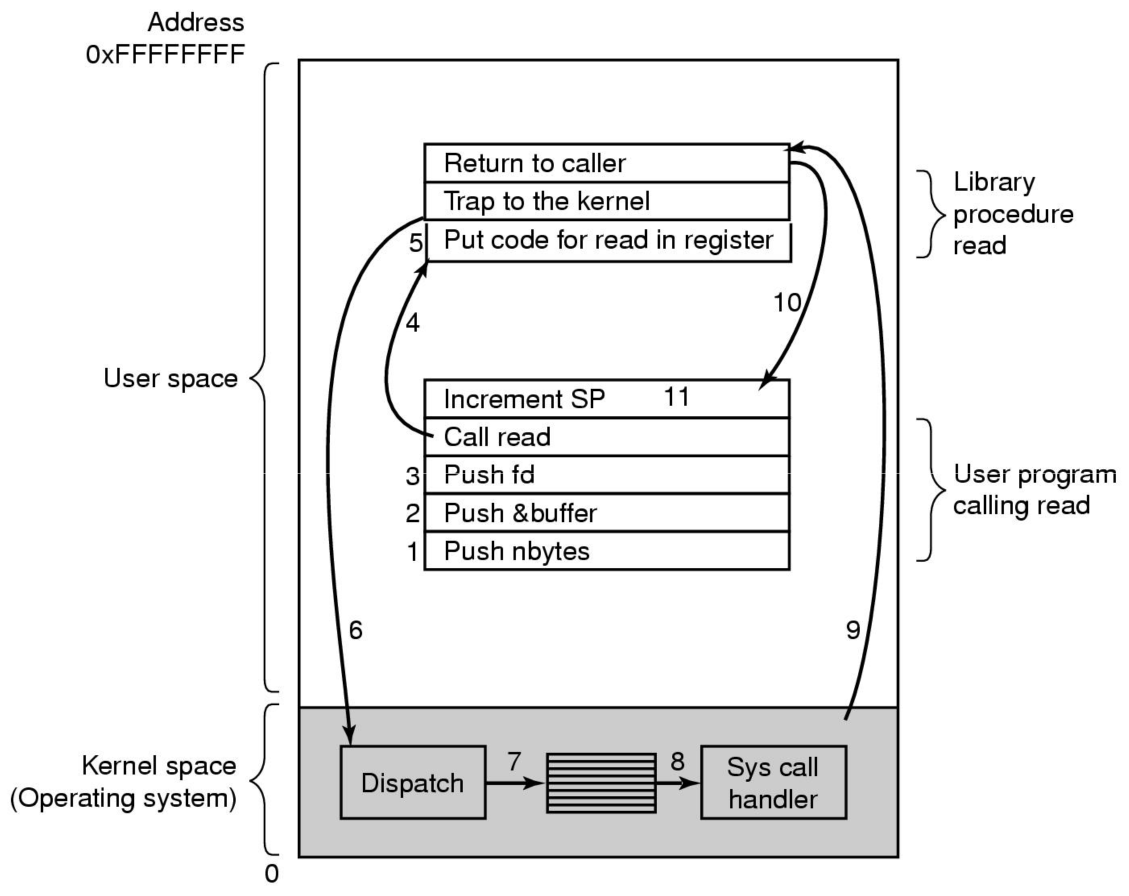
Different kinds of shells

	sh	csh	ksh	bash	tcsh	zsh
Job control	N	Y	Y	Y	Y	Y
Aliases	N	Y	Y	Y	Y	Y
Shell functions	Y	N	Y	Y	N	Y
"Sensible" Input/Output redirection	Y	N	Y	Y	N	Y
Directory stack	N	Y	Y	Y	Y	Y
Command history	N	Y	Y	Y	Y	Y
Command line editing	N	N	Y	Y	Y	Y
Vi Command line editing	N	N	Y	Y	Y	Y
Emacs Command line editing	N	N	Y	Y	Y	Y
Rebindable Command line editing	N	N	N	Y	Y	Y
User name look up	N	Y	Y	Y	Y	Y
Login/Logout watching	N	N	N	N	Y	Y
Filename completion	N	Y	Y	Y	Y	Y
Username completion	N	Y	Y	Y	Y	Y
Hostname completion	N	Y	Y	Y	Y	Y

Y – Yes
N – No
F - shells function mechanism.

http://www.faqs.org/faqs/unix-faq/shell/shell-differences/

Steps in Making a System Call



There are 11 steps in making the system call
read (fd, buffer, nbytes)

Some System Calls For Process Management

Process management

Call	Description
pid = fork( )	Create a child process identical to the parent
pid = waitpid(pid, &statloc, options)	Wait for a child to terminate
s = execve(name, argv, environp)	Replace a process' core image
exit(status)	Terminate process execution and return status

Some System Calls For File Management

File management

Call	Description
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
position = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file’s status information

Some System Calls For Directory Management

Directory and file system management

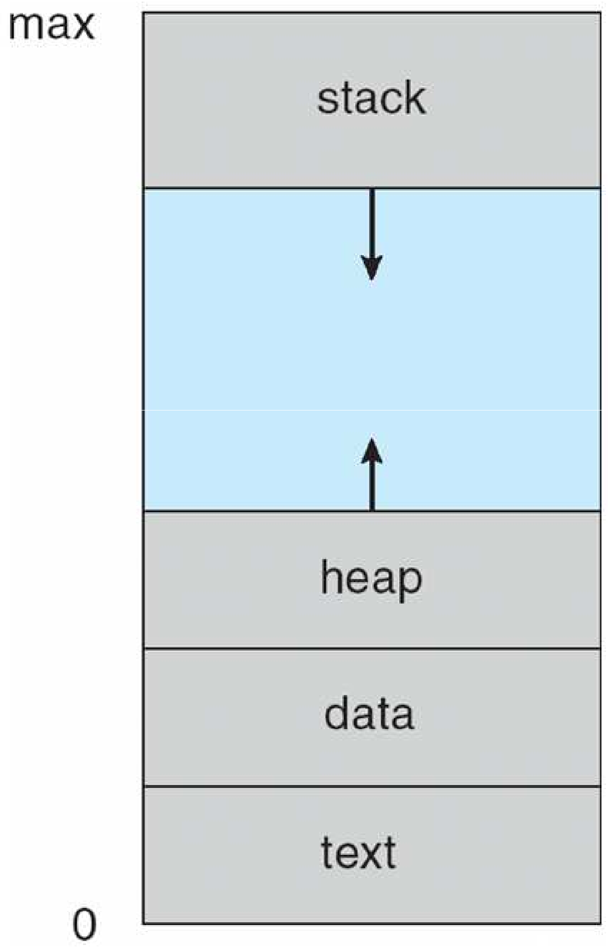
Call	Description
s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system

Some System Calls For Miscellaneous Tasks

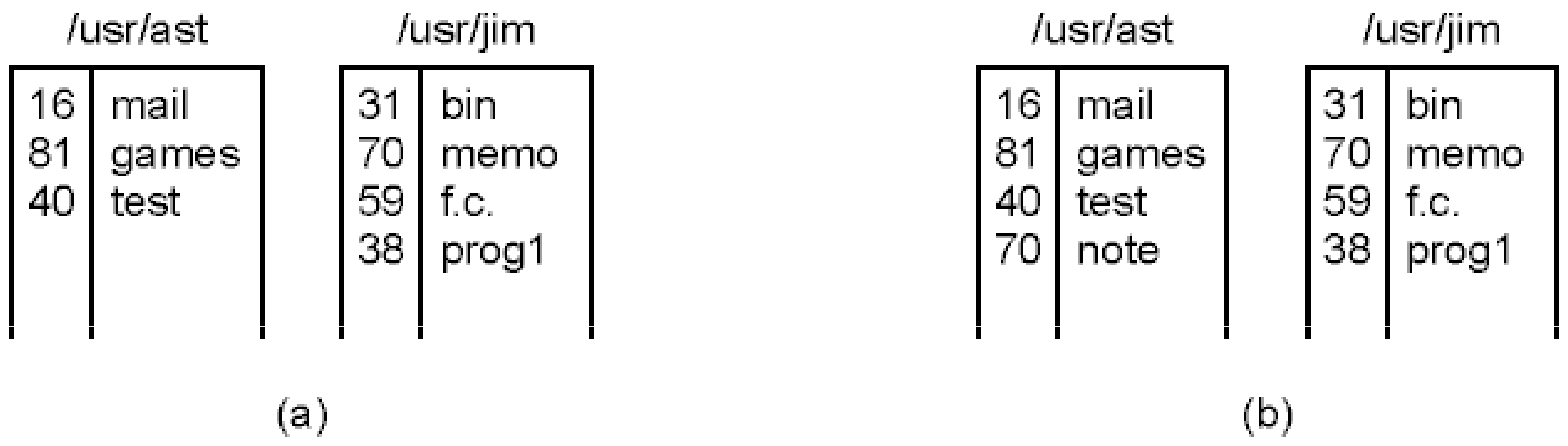
Miscellaneous

Call	Description
s = chdir(dirname)	Change the working directory
s = chmod(name, mode)	Change a file's protection bits
s = kill(pid, signal)	Send a signal to a process
seconds = time(&seconds)	Get the elapsed time since Jan. 1, 1970

System Calls (2)



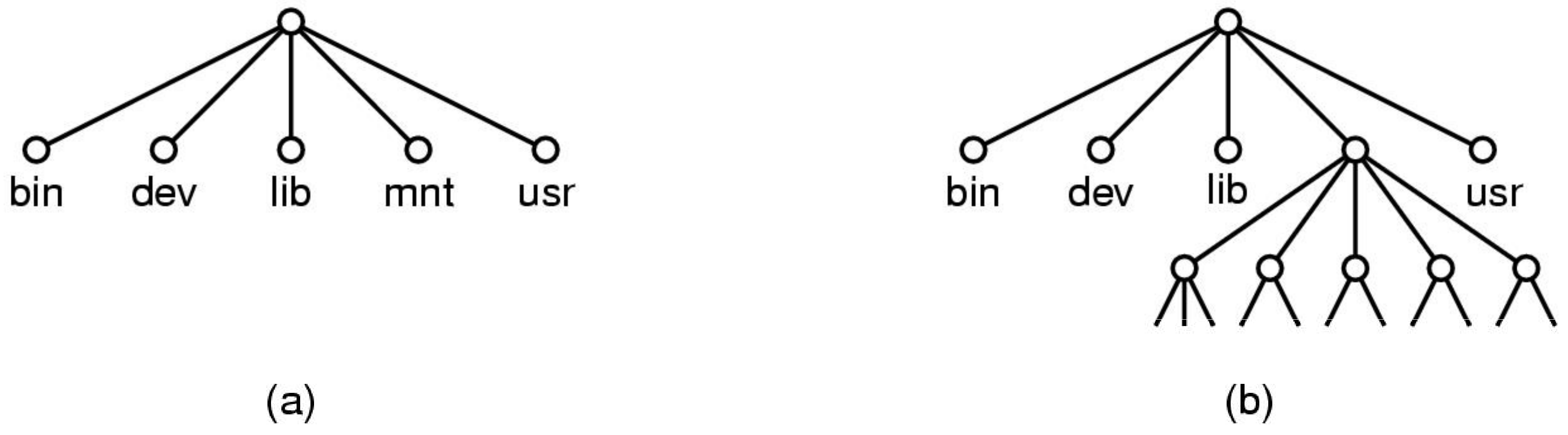
System Calls (3)



(a) Two directories before linking *usr/jim/memo* to ast's directory

(b) The same directories after linking

System Calls (4)



(a) File system before the mount

(b) File system after the mount

System Calls (5)

UNIX	Win32	Description
fork	CreateProcess	Create a new process
waitpid	WaitForSingleObject	Can wait for a process to exit
execve	(none)	CreateProcess = fork + execve
exit	ExitProcess	Terminate execution
open	CreateFile	Create a file or open an existing file
close	CloseHandle	Close a file
read	ReadFile	Read data from a file
write	WriteFile	Write data to a file
lseek	SetFilePointer	Move the file pointer
stat	GetFileAttributesEx	Get various file attributes
mkdir	CreateDirectory	Create a new directory
rmdir	RemoveDirectory	Remove an empty directory
link	(none)	Win32 does not support links
unlink	DeleteFile	Destroy an existing file
mount	(none)	Win32 does not support mount
umount	(none)	Win32 does not support mount
chdir	SetCurrentDirectory	Change the current working directory
chmod	(none)	Win32 does not support security (although NT does)
kill	(none)	Win32 does not support signals
time	GetLocalTime	Get the current time

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

Concurrency

```
$ ls –la
```

Shell creates a process to run this command

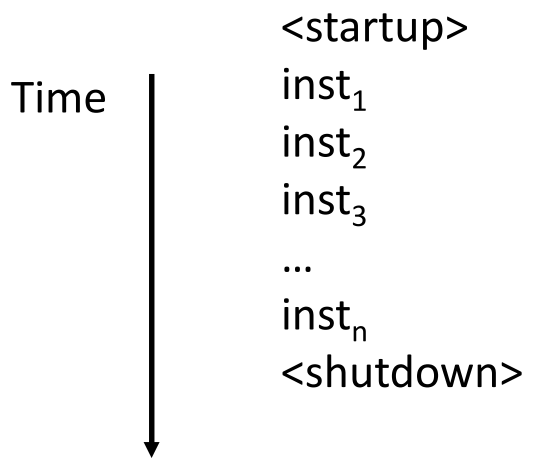
```
$ ls –la | more
```

- 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



<startup>
$inst_1$
$inst_2$
$inst_3$
…
$inst_n$
<shutdown>

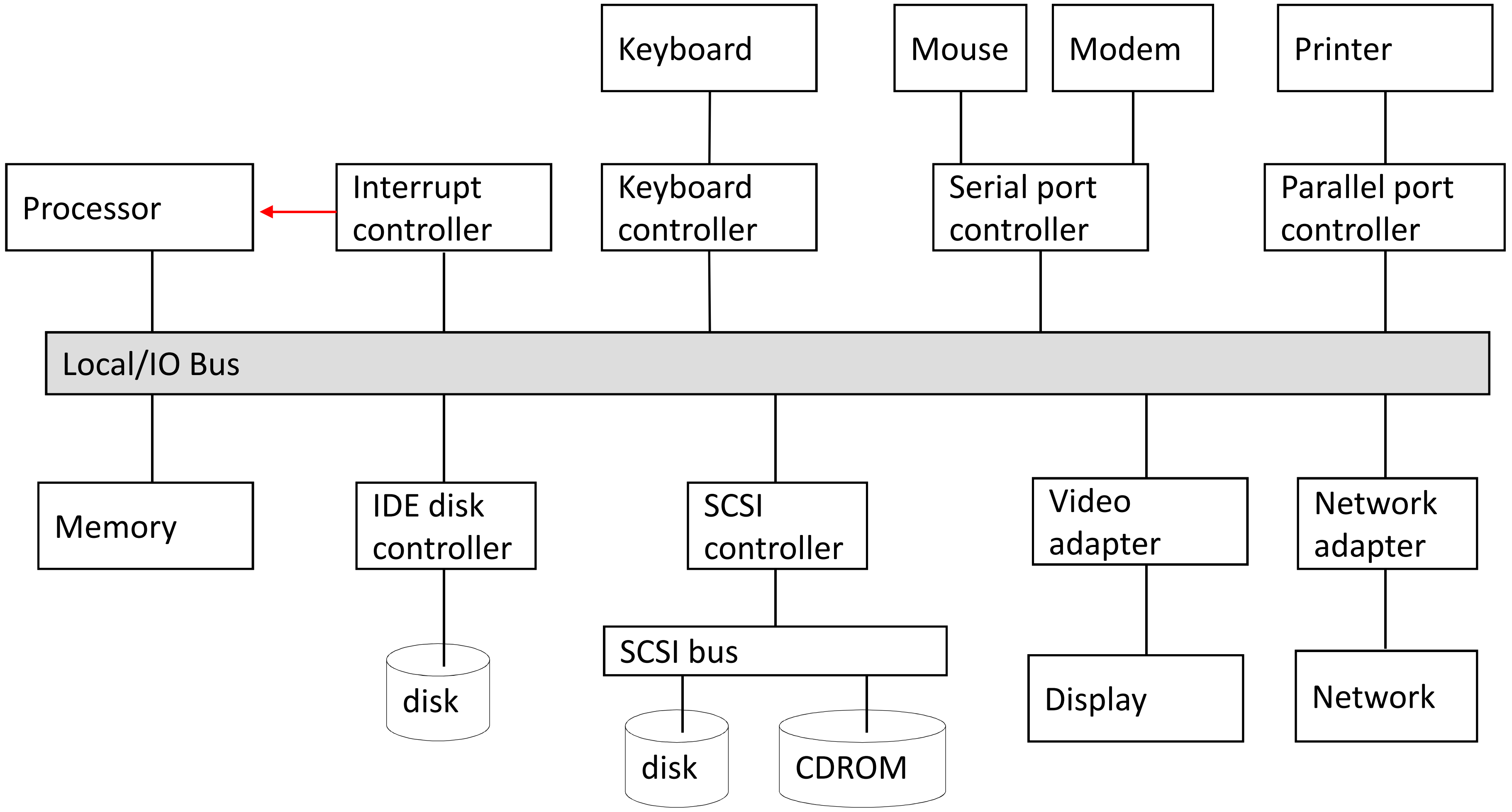
Altering the Control Flow

- 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

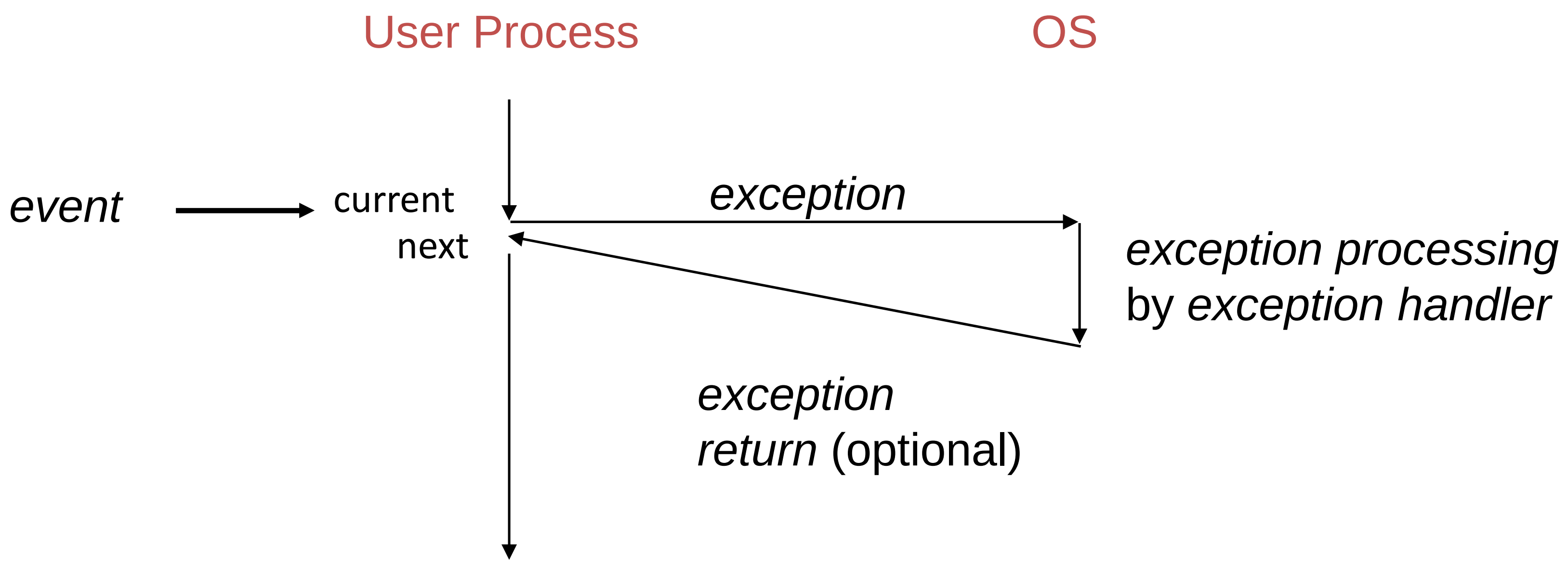
  - Mechanisms for exceptional control flow exists at all levels of a computer system.
- Low level Mechanism
  - exceptions
    - change in control flow in response to a system event (i.e., change in system state)
  - Combination of hardware and OS software
- Higher Level Mechanisms
  - Process context 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

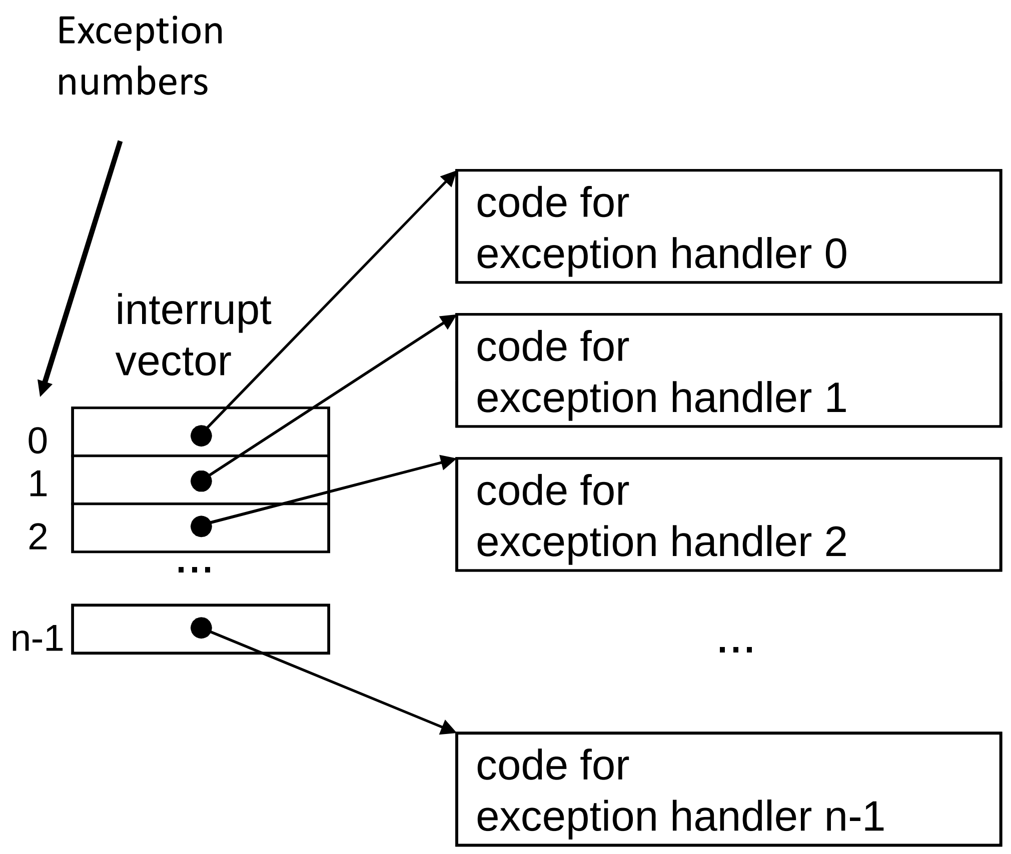


Exceptions

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



Interrupt Vectors



Exception numbers
interrupt vector
0
1
2
...
n-1
code for exception handler 0
code for exception handler 1
code for exception handler 2
...
code for exception handler n-1

- 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)

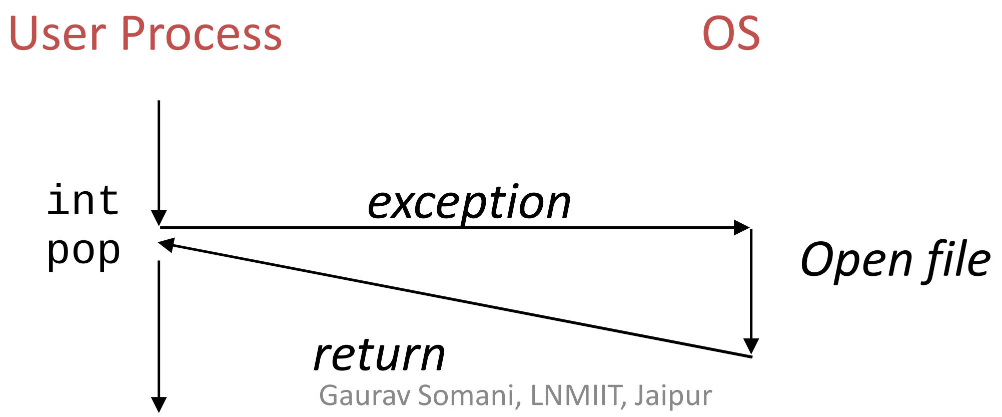
- Caused by events external to the processor
  - Indicated by setting the processor’s interrupt pin
  - handler returns to “next” instruction.
- Examples:
  - I/O interrupts
    - hitting ctl-c at the keyboard
    - arrival of a packet from a network
    - arrival of a data sector from a disk
  - Hard reset interrupt
    - hitting the reset button on a uC
  - 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)`
    - Function open executes system call instruction int
  - OS must find or create file, get it ready for reading or writing
  - Returns integer file descriptor

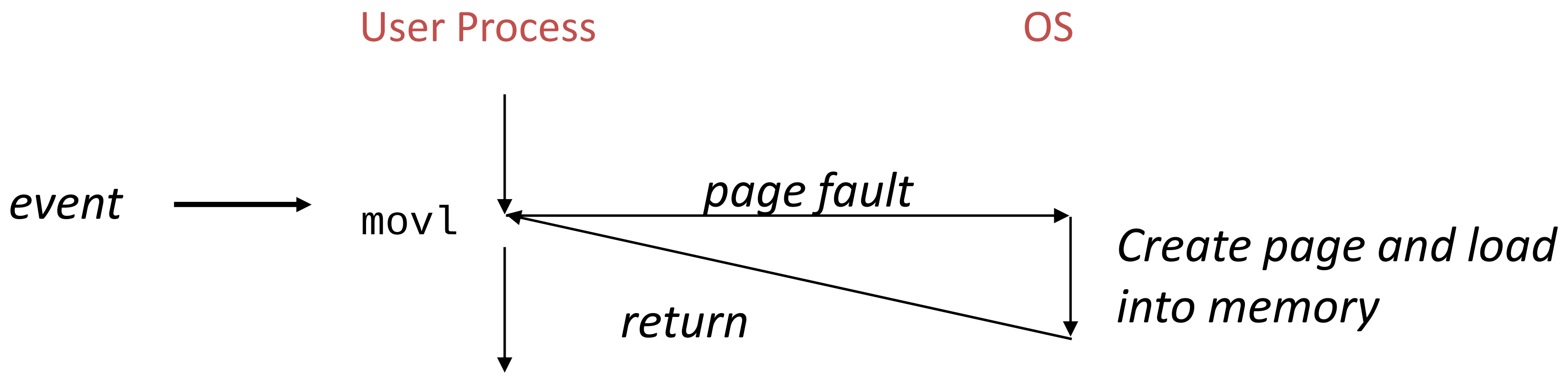


Fault Example #1

- 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;
}
```

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

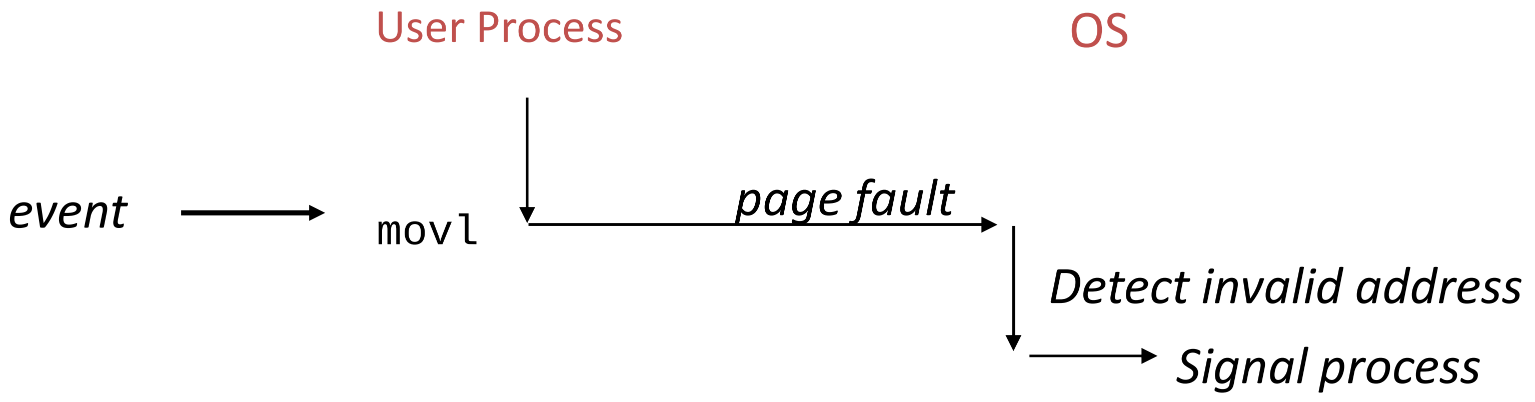


Fault Example #2

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

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

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



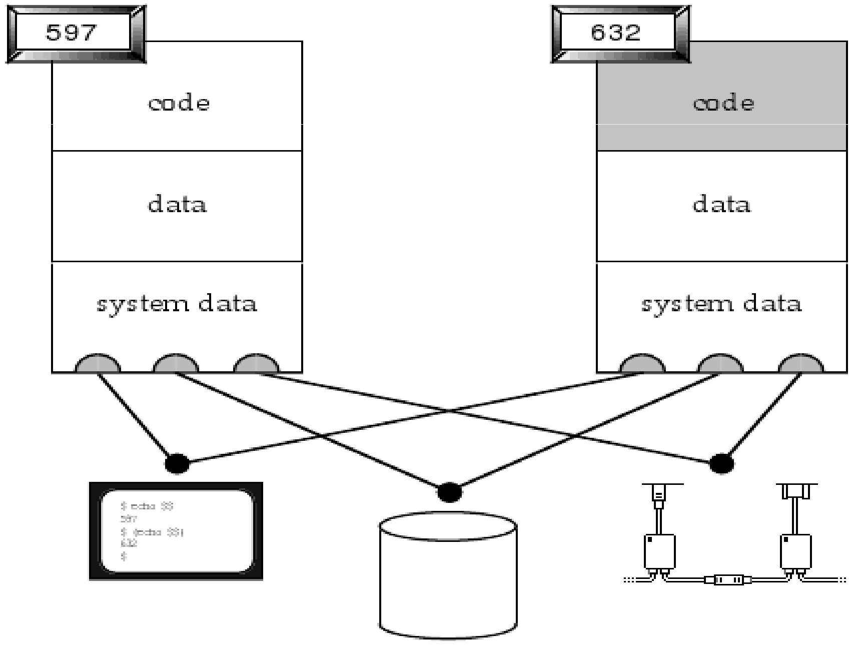
System Calls for Process Creation and Manipulation

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

Forking

UNIX 'fork' duplicates process:

- copies complete process state:
  - program data + system data
  - including file descriptors
- code immutable – shared



```
$ echo $$
597
$ (echo $$)
632
$
```

Forking

- old process called the <u>parent</u>
- new process called the <u>child</u>
- process ids
  - allocated sequentially
  - so effectively unique
    (but do wrap after a very long time)
- finding process ids
  - at the shell prompt:
    use '**ps**'
  - in a C program:
    use '**int p = getpid();**'
  - in a shell script:
    use '**$$**'

  N.B. useful for naming temporary files:

  ```
  tmpfile = "/tmp/myfile$$"
  ```

Fork System Call

```
pid_t  p  =  fork();
```

(pid_t ≈ int)

- if successful
  - process
  - successful fork returns:

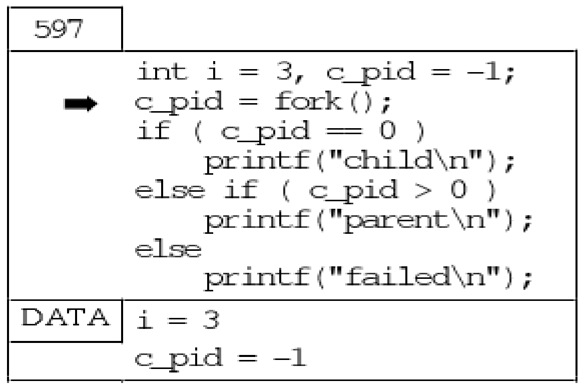
0	–	to child process
child pid	–	to parent process

  $\Rightarrow$ parent and child are different!

- negative result on failure

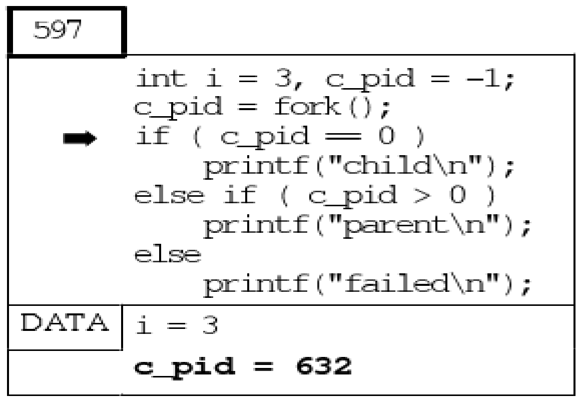
Example

- parent forks

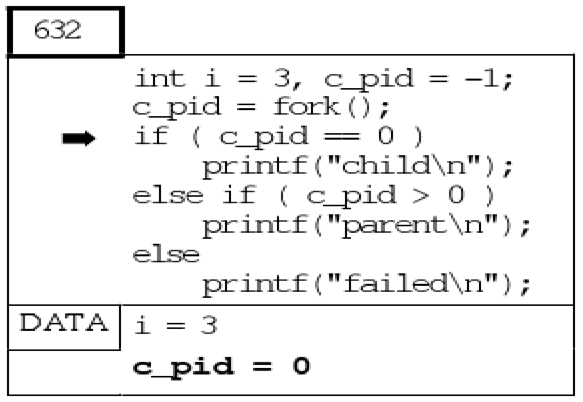


```
597
      int i = 3, c_pid = -1;
 ➡    c_pid = fork();
      if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = -1
```

- after fork parent and child identical



```
597
      int i = 3, c_pid = -1;
      c_pid = fork();
 ➡    if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = 632
```

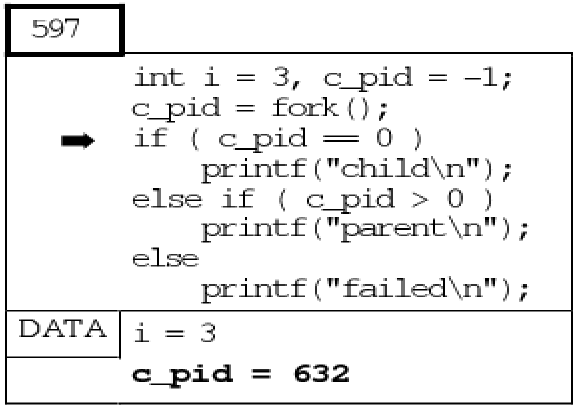


```
632
      int i = 3, c_pid = -1;
      c_pid = fork();
 ➡    if ( c_pid == 0 )
          printf("child\n");
      else if ( c_pid > 0 )
          printf("parent\n");
      else
          printf("failed\n");
DATA  i = 3
      c_pid = 0
```

- except for the return value of fork

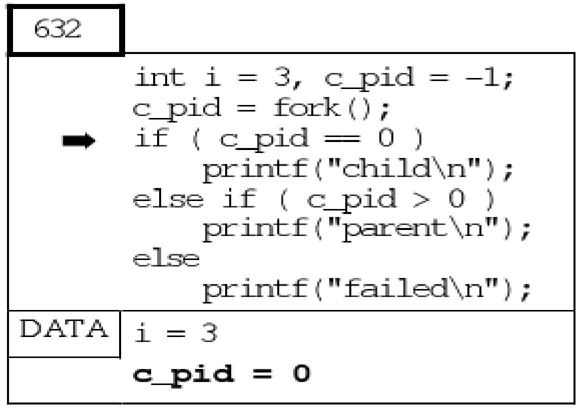
Example

- because data are different



```
int i = 3, c_pid = -1;
c_pid = fork();
if ( c_pid == 0 )
    printf("child\n");
else if ( c_pid > 0 )
    printf("parent\n");
else
    printf("failed\n");
```

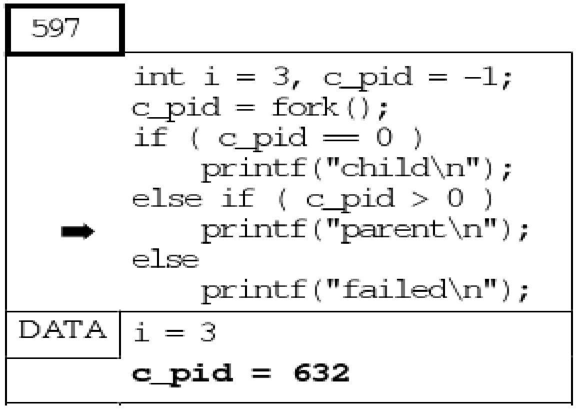
```
i = 3
c_pid = 632
```



```
int i = 3, c_pid = -1;
c_pid = fork();
if ( c_pid == 0 )
    printf("child\n");
else if ( c_pid > 0 )
    printf("parent\n");
else
    printf("failed\n");
```

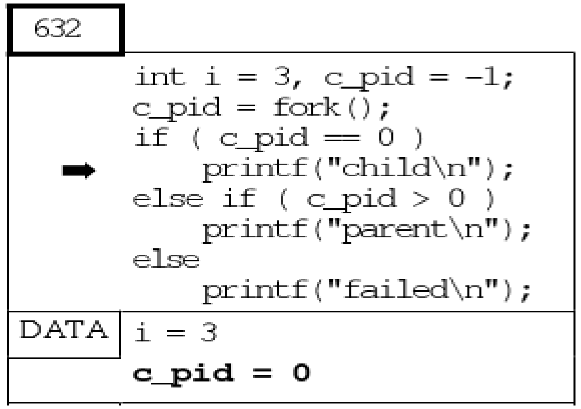
```
i = 3
c_pid = 0
```

- program execution differs



```
int i = 3, c_pid = -1;
c_pid = fork();
if ( c_pid == 0 )
    printf("child\n");
else if ( c_pid > 0 )
    printf("parent\n");
else
    printf("failed\n");
```

```
i = 3
c_pid = 632
```



```
int i = 3, c_pid = -1;
c_pid = fork();
if ( c_pid == 0 )
    printf("child\n");
else if ( c_pid > 0 )
    printf("parent\n");
else
    printf("failed\n");
```

```
i = 3
c_pid = 0
```

- so parent and child behaviour diverge

Process Control Block

- PCB, also called Task Control Block or Task Structure is a data structure in the operating system kernel containing the information needed to manage a particular process.

Included information

- The identifier of the process (a process identifier, or **PID**)
- Register values for the process including, notably, the **program counter** and **stack pointer** values for the process.
- The **address space** for the process
- **Priority** (in which higher priority process gets first preference. eg., **nice** value on Unix operating systems)
- **Process accounting** information, such as when the process was last run, how much CPU time it has accumulated, etc.
- **Pointer to the next PCB** i.e. pointer to the PCB of the next process to run
- **I/O Information** (i.e. I/O devices allocated to this process, list of opened files, etc)

Process Creation

- Example of fork() programs

```
#include <sys/types.h>
#include <unistd.h> /* for fork() */
#include <stdio.h>
main()
{
pid_t pid;      /*holds process-id in parent*/
printf("One\n");
pid=fork();
printf("Two\n");
}
[root@localhost programs]# gcc -o fork fork.c
[root@localhost programs]# ./fork
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

The specific attributes of the child process that differ from the parent process are

- The child process has its own unique process ID.
- The parent process ID of the child process is the process ID of its parent process.
- The child process gets its own copies of the parent process's open file descriptors. Subsequently changing attributes of the file descriptors in the parent process won't affect the file descriptors in the child, and vice versa. However, both processes share the file position associated with each descriptor.
- The elapsed processor times for the child process are set to zero.
- The child doesn't inherit file locks set by the parent process.
- The child doesn't inherit alarms set by the parent process.
- The set of pending signals for the child process is cleared.

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.

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.

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?

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 System Call

```
execv(char *prog, char **argv);
```

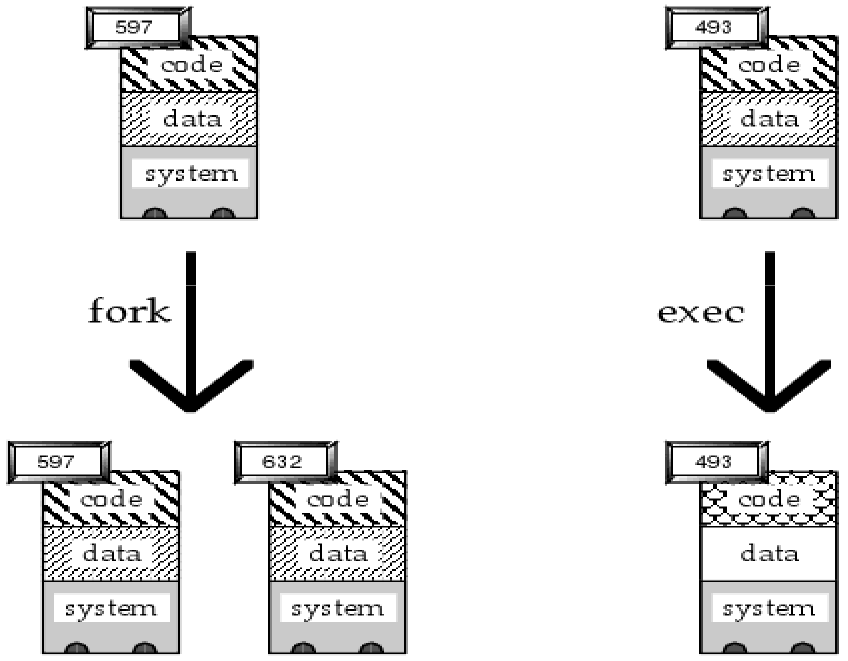
- <u>replaces</u> the current process with `prog`
- never returns except on failure
- `argv` is passed to the '`main`' of `prog`
  N.B. needs at least `argv[0]` set to program name
- new process:

code	–	replaced by `prog`
data	–	reinitialised
system data	–	partly retained

  ✳ file descriptors still open
- several variants (`execl`, `execvp`, ...)
- often used after `fork` to spawn a fresh program

Exec vs. Fork

- fork <u>duplicates</u> process
- exec <u>replaces</u> process



- fork child shares open file descriptors
- exec-ed process retains open fds

exec family tree

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

$$\begin{tikzcd}
\text{execl} \arrow[d] & \text{execle} \arrow[dd] & \text{execlp} \arrow[d] \\
\text{execv} \arrow[dr] & & \text{execvp} \arrow[dl] \\
& \text{execve} &
\end{tikzcd}$$

The real system call

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 program 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.
  - As with the execl, there is an execp() from that uses the search path.
  - The arguments to exec form the arguments to main() of the executed program.

When will you create processes?

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.

exec.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.

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.

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;
```

exec_wait.c

```
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 Chil
      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);
}
```

```
linux-ayhj:~ # ./exec_wait
PID before fork, ie PID of current process: 12929
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: 12930
The value of Parent PID, who created me: 12929
OUTPUT OF LS GOES HERE
Message from a parent process
PID of child known to the parent: 12930
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.

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>. 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

- 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.

Death of a Process

- when parent dies
  - children become orphans !
  - system init process 'adopts' them

- when child dies
  - parent (or init) informed by signal
    (`SIGCHLD`)
  - child process partly destroyed
  - rump retained until parent 'reaps'
    - using `wait` or `wait3` system call
  - until then child is 'zombie'
    - `ps` says `<exiting>` or `<defunct>`

N.B. zombie state necessary so parent can discover <u>which</u> child died

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.

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.

/proc

- /proc/*PID*/cmdline, which contains the command which originally started the process.
- /proc/*PID*/cwd, a symlink to the current working directory of the process.
- /proc/*PID*/environ, a file containing the names and contents of the environment variables that affect the process.
- /proc/*PID*/exe, a symlink to the original executable file, if it still exists (a process may continue running after its original executable has been deleted or replaced).
- /proc/*PID*/fd, a directory containing a symbolic link for each open file descriptor.
- /proc/*PID*/root, a symlink to the root path as seen by the process. For most processes this will be a link to / unless the process is running in a chroot jail.
- /proc/*PID*/status, a file containing basic information about a process including its run state and memory usage.
- /proc/*PID*/task, a directory containing hard links to any tasks that have been started by this (i.e.: the parent) process.
- /proc/*PID*/maps, the memory map showing which addresses currently visible to that process are mapped to which regions in RAM or to files.

/proc

- /proc/cpuinfo
- /proc/devices
- /proc/filesystems
- /proc/interrupts
- /proc/iomem
- /proc/ioports
- /proc/irq
- /proc/mounts
- /proc/net
- /proc/partitions
- /proc/swaps
- /proc/version

Tools to manipulate processes (cont) STRACE……….

```
linux-ayhj:~ # cat a.c
#include <stdio.h>
int main()
{
printf("Hi");
return 0;
}

linux-ayhj:~ # gcc –static –o hi a.c
linux-ayhj:~ # strace ./hi
execve("./hi", ["./hi"], [/* 61 vars */]) = 0
uname({sys="Linux", node="linux-ayhj", ...}) = 0
brk(0)                                  = 0x80cf000
brk(0x80cfcd0)                          = 0x80cfcd0
set_thread_area({entry_number:-1 -> 6, base_addr:0x80cf830,
   limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
   limit_in_pages:1, seg_not_present:0, useable:1}) = 0
brk(0x80f0cd0)                          = 0x80f0cd0
brk(0x80f1000)                          = 0x80f1000
fstat64(1, {st_mode=S_IFCHR|0620, st_rdev=makedev(136, 1), ...}) = 0
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1,
   0) = 0xb7897000
write(1, "Hi", 2Hi)                       = 2
exit_group(0)                           = ?
linux-ayhj:~ #
```

Thanks

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.