CPS 310
Unix Process Model

Jeff Chase
Duke University

http://www.cs.duke.edu/~chase/cps310
Programs run as independent processes.

Protected OS kernel mediates access to shared resources.

Each process has a private virtual address space and one or more threads.

Protected system calls

...and upcalls (e.g., signals)

Threads enter the kernel for OS services.

The kernel code and data are protected from untrusted processes.
Today

• We understand threads and concurrency now.
• Add a kernel and address spaces → “vanilla Unix”
• Flesh out Unix process model and syscall API
• Kernel interactions
• Using the process model
• **Imagine:**
  – Your thread library is “really” a kernel.
  – Each thread is a classical process.
  – i.e., Each thread has its own VAS.
  – i.e., Threads/processes share data only within/via the kernel.
Thread states and transitions

We will presume that these transitions occur only in kernel mode. This is true in classical Unix and in systems with pure kernel-based threads.

Before a thread can **sleep**, it must first enter the kernel via trap (syscall) or fault.

Before a thread can **yield**, it must enter the kernel, or the core must take an interrupt to return control to the kernel.

On entry to the **running** state, kernel code decides if/when/how to enter user mode, and sets up a suitable context. E.g., for initial start, return from fault or syscall, or to deliver a signal.
Threads execute user code on a **user stack** in user space (the process virtual address space).

Each thread has a second **kernel stack** in **kernel space** (VM accessible only in kernel mode).

System calls and faults run in kernel mode on a kernel stack for the current thread.

Kernel code running in P’s process context has access to P’s virtual memory.

The syscall (trap) handler makes an indirect call through the system call dispatch table to the handler registered for the specific system call.
Upcall example: Unix signals

- Signals are asynchronous notifications to a process that some event of interest to it has occurred.
- A process may register **signal handlers** for various events relating to the process. The signal handlers are procedures in user space.
- To deliver a signal, the kernel redirects a user thread to execute a selected registered signal handler in user mode.
- Unix signals take a default action if no handler is registered.
  - E.g., segmentation fault → die. Other actions: ignore, stop

Protected system calls

...and upcalls (e.g., signals)
The kernel must be bulletproof

Secure kernels handle system calls verrry carefully.

Syscalls indirect through syscall dispatch table by syscall number. No direct calls to kernel routines from user space!

What about references to kernel data objects passed as syscall arguments (e.g., file to read or write)?

Use an integer index into a kernel table that points at the data object. The value is called a handle or descriptor. No direct pointers to kernel data from user space!

Kernel copies all arguments into kernel space and validates them.

Kernel interprets pointer arguments in context of the user VAS, and copies the data in/out of kernel space (e.g., for read and write syscalls).
Unix: A lasting achievement?

“Perhaps the most important achievement of Unix is to demonstrate that a powerful operating system for interactive use need not be expensive… it can run on hardware costing as little as $40,000.”

The UNIX Time-Sharing System*
D. M. Ritchie and K. Thompson
1974

DEC PDP-11/24

http://histoire.info.online.fr/pdp11.html
Process management

• OS offers system call APIs for managing processes.
  – Create processes (children)
  – Control processes
  – Monitor process execution
  – “Join”: wait for a process to exit and return a result
  – “Kill”: send a signal to a process
  – Establish interprocess communication (IPC: later)
  – Launch a program within a process

• We study the Unix process abstraction as an example.
  – Illustrative and widely used for 40+ years!
  – Optional: Use it to build your own shell.
Example: Unix fork

The Unix **fork()** system call creates/launches a new thread, in its own fresh virtual address space: it creates a new process. (Thread + VAS == Process.)

Strangely, the new ("child") process is an exact clone of the calling ("parent") process.
Unix fork/exit syscalls

- **fork()**: Create a new process that is a clone of its parent. Return child process ID (pid) to parent, return 0 to child.

- **exit(status)**: Exit with status, destroying the process. Status is returned to the parent.

**Note**: this is not the only way for a process to exit!
The `fork` syscall returns twice:

It returns a zero in the context of the new child process.

It returns the new child process ID (pid) in the context of the parent.

```c
int pid;
int status = 0;
if (pid = fork()) {
    /* parent */
    ..... 
} else {
    /* child */
    ..... 
    exit(status);
}
```
A simple program: sixforks

... int main(int argc, char* argv) {
    fork();
    fork();
    fork();
    fork();
    fork();
    fork();
    printf("Process %d exiting.\n", getpid());
}  

getpid syscall: 
Get processID of current process.

chase$ cc -o sixforks sixforks.c
chase$ ./sixforks
???
chase$
A simple program: sixforks

... 
int
main(int argc, char* argv)
{
    fork();
    fork();
    fork();
    fork();
    printf("Process %d exiting.\n", getpid());
}
sixforks: some questions

- What if I want to create six children, but I don’t want my children to have children of their own?
- What if I want the program to print the total number of processes created? How? (Other than by having the program do the math.)
- How much memory does this program use? How many pages?
- How does this test system assign process IDs?
- Why do the process IDs print out of order?
fork (original concept)

5.1 Processes

Except while UNIX is bootstrapping itself into operation, a new process can come into existence only by use of the fork system call:

processid = fork (label)

When *fork* is executed by a process, it splits into two independently executing processes. The two processes have independent copies of the original core image, and share any open files. The new processes differ only in that one is considered the parent process: in the parent, control returns directly from the *fork*, while in the child, control is passed to location *label*. The *processid* returned by the *fork* call is the identification of the other process.

Because the return points in the parent and child process are not the same, each image existing after a *fork* may determine whether it is the parent or child process.
fork in action today

void
dofork()
{
  int cpid = fork();
  if (cpid < 0) {
    perror("fork failed: ");
    exit(1);
  } else if (cpid == 0) {
    child();
  } else {
    parent(cpid);
  }
}

Fork is conceptually difficult but syntactically clean and simple.

I don’t have to say anything about what the new child process “looks like”: it is an exact clone of the parent!

The child has a new thread executing at the same point in the same program. The child is a new instance of the running program: it has a “copy” of the entire address space. The “only” change is the process ID and return value cpid!

The parent thread continues on its way. The child thread continues on its way.
wait syscall

```c
int pid;
int status = 0;
if (pid = fork()) {
    /* parent */
    ...
    pid = wait(&status);
} else {
    /* child */
    ...
    exit(status);
}
```

**Warning:** the uses of the terms **wait** and **signal** should not be confused with the monitor/CV primitives of the same names.

Parent uses **wait** to sleep until the child exits; **wait** returns child pid and status.

**Wait** variants allow wait on a specific child, or notification of stops and other signals. **Recommended**: use **waitpid()**.
wait

Process states (i.e., states of the main thread of the process)

Note: in modern Unix systems the `wait` syscall has many variants and options.

5.4 Process Synchronization
Another process control system call

```c
processid = wait( )
```
causes its caller to suspend execution until one of its children has completed execution. Then `wait` returns the `processid` of the terminated process. An error return is taken if the calling process has no descendants. Certain status from the child process is also available. `Wait` may also present status from a grandchild or more distant ancestor; see §5.5.

5.5 Termination
Lastly,

```c
exit (status)
```
terminates a process, destroys its image, closes its open files, and generally obliterates it. When the parent is notified through the `wait` primitive, the indicated `status` is available to the parent; if the parent has already terminated, the status is available to the grandparent, and so on. Processes
A simple program: forkdeep

```c
int count = 0;
int level = 0;

void child() {
    level++;
    output pids
    if (level < count)
        dofork();
    if (level == count)
        sleep(3); /* pause 3 secs */
}

void parent(int childpid) {
    output pids
    wait for child to finish
}

main(int argc, char *argv[]) {
    count = atoi(argv[1]);
    dofork();
    output pid
}

We’ll see later where arguments come from.
```

level==1

level==2
```
chase$ ./forkdeep 4
  30866-> 30867
    30867
  30867-> 30868
    30868
  30868-> 30869
    30869
  30869-> 30870
    30870
  30870
    30869
  30869
    30868
  30868
    30867
  30867
  30866
chase$
```

```
chase$ ./forkdeep 3
  11496-> 11498
    11498
  11498-> 11499
    11499
  11499-> 11500
    11500
    11500
    11499
    11498
  11496
chase$
```
Example: Chrome browser

When we started this project, the guys were saying that one of the problems with browsers is that they're inherently single-threaded.

For example, once you have JavaScript executing, it's going to keep going, and the browser can't do anything else until JavaScript returns control to the browser.
Processes in the browser

Chrome makes an interesting choice here. But why use processes?
Problem: heap memory and fragmentation

But as time goes on, fragmentation results -- little bits of memory still get used even when a tab gets closed.

Either we have memory that nothing can refer to again, or there's a piece of de-allocated memory we still have pointers to.

So when the browser wants to open a new tab, it can't fit it in the existing space --

[Google Chrome Comics]
Solution: whack the whole process

When a process exits, all of its virtual memory is reclaimed as one big slab.

[Google Chrome Comics]
Processes for fault isolation

When a plugin combines with HTML and JavaScript, it all runs in the same process.

If any part crashes or starts corrupting memory, they're all hosed.

So I worked on ripping plugins out of the rendering process and putting them in a separate process all their own.
WITH SANDBOXING, OUR GOAL IS TO PREVENT MALWARE FROM INSTALLING ITSELF ON YOUR COMPUTER OR USING WHAT HAPPENS IN ONE TAB TO AFFECT WHAT HAPPENS IN ANOTHER.

SO, FOR EACH OF THESE PROCESSES WE'VE STRIPPED AWAY ALL OF THEIR RIGHTS.

THEY CAN COMPUTE BUT THEY CAN'T WRITE FILES TO YOUR HARD DRIVE OR READ FILES FROM SENSITIVE AREAS LIKE YOUR DOCUMENTS OR DESKTOP.
Processes: A Closer Look

virtual address space + thread(s) + process control block (PCB)

The address space is a private name space for a set of memory segments used by the process.

The kernel must initialize the process virtual memory for the program to run.

Each process has at least one thread (the “main thread”) bound to the VAS.

Each thread has a stack addressable in the VAS.

The kernel can suspend/restart a thread wherever and whenever it wants.

The OS maintains some kernel state for each process in the kernel’s internal data structures: e.g., a file descriptor table, links to maintain the process tree, current directory, and a place to store the exit status.
Process, kernel, and syscalls

syscall stub
syscall dispatch table
I/O descriptor table
I/O objects

process user space
user buffers
read() {
...
}

trap

write() {
...
}

copyout
copyin

syscall dispatch table
I/O objects

read() {
...
}

return to user mode
Unix fork/exec/exit/wait syscalls

```c
int pid = fork();
Create a new process that is a clone of its parent, running the same program.

eexec("program" [argvp, envp]);
Overlay the calling process with a new program, and transfer control to it, passing arguments and environment.

exit(status);
Exit with status, destroying the process.

int pid = wait(&status);
Wait for exit (or other status change) of a child, and “reap” its exit status.
Recommended: use waitpid().
```
But how do I run a new program in my child process?

- The child, or any process really, can replace its program in midstream.
- `exec*` system call: “forget everything in my address space and reinitialize my entire address space with stuff from a named program file.”
- The exec system call never returns: the new program executes in the calling process until it dies (exits).
  - The code from the parent program runs in the child process and controls its future. The parent program selects the child program that the child process will run (via `exec`)
  - But don’t forget to check error status from exec*! It returns an error to parent program if it fails.
When a program launches, the OS creates a **process** to run it, with a main **thread** to execute the code, and a **virtual memory** to store the running program’s code and data.
5.3 Execution of Programs

Another major system primitive is invoked by

\texttt{execute(file, arg_1, arg_2, ..., arg_n)}

which requests the system to read in and execute the program named by \texttt{file}, passing it string arguments \texttt{arg_1, arg_2, ..., arg_n}. Ordinarily, \texttt{arg_1} should be the same string as \texttt{file}, so that the program may determine the name by which it was invoked. All the code and data in the process using \texttt{execute} is replaced from the file, but open files, current directory, and interprocess relationships are unaltered. Only if the call fails, for example because \texttt{file} could not be found or because its execute-permission bit was not set, does a return take place from the \texttt{execute} primitive; it resembles a “jump” machine instruction rather than a subroutine call.
A simple program: `forkexec`

```c
main(int argc, char *argv[]) {
    int status;
    int rc = fork();
    if (rc < 0) {
        perror("fork failed: ");
        exit(1);
    } else if (rc == 0) {
        printf("I am a child: %d.\n", getpid());
        argv++;
        execve(argv[0], argv, 0);
    } else if (rc == 0) {
        printf("I am a child: %d.\n", getpid());
        argv++;
        execve(argv[0], argv, 0);
        /* NOTREACHED */
    } else {
        waitpid(rc, &status, 0);
        printf("Child %d exited with status %d\n.\n", rc,
               WEXITSTATUS(status));
    }
}
```

Always check return from syscalls and show any errors!

Parent program running in child process

A successful `exec*` never returns to calling program.

Reap `exit status` return value from child via `exit/wait`. 

A simple program: prog0

```c
... int main() {
    printf("Hi from %d!\n", getpid());
    exit(72);
}
```

*getpid* syscall: Get processID of current process.

*exit* syscall: Pass *exit status* return value to parent via exit/wait.

chase$ cc –o forkexec forkexec.c
chase$ cc –o prog0 prog0.c
chase$ ./forkexec prog0
I am a child: 11384.
Hi from 11384!
Child 11384 exited with status 72.
chase$
Kernel/user transitions for fork/exec/exit

The kernel may start and kill threads, and/or arbitrarily change the user virtual memory and/or thread context.

It does it all the time.
About the previous slide

• **A trap** is a system call, e.g., fork/exec/exit/wait/ or open/close/read/write or pipe/dup2.

• **A program** is an executable file that may be launched in a process, e.g., with an exec* system call. When a program is running in a process that program controls the process. More precisely, the program controls the behavior of a thread in the process while that thread is running in user mode.

• When I say that "a program invokes a system call" or "a process does a trap" I mean that a thread is running in a user program in a process, and that thread executes a trap instruction in the program, for the purpose of entering the kernel to perform a system call.

In the example:

• Exec* system call is invoked by the **parent program** running in the **child process**.

• Exec* system call "returns" into the program whose name was the first argument to exec*. That is the program I call the **child program**: it is now running in the child process, having replaced the parent program in the child process. After exec*, the child program begins executing in its main(). (Be sure you understand how that happened.)
What does this code do?

```c
int main(int argc, char *argv[]) {
    printf("about to run program %s.\n", argv[0]);
    execve(argv[0], argv, 0);
    perror("exec failed");
}
```
The details aren’t important. The point is:

The exec system call sets up the VAS of the calling process to execute a named program.

Exec passes two arrays of strings to the new program’s main(): an array of arguments and an array of named environment variables.

It stages the argv/env arrays in the VAS before returning to user mode to start execution at main().
But how is the first process made?
Init and Descendants

Kernel “handcrafts” initial process to run “init” program.

Other processes descend from init, including one instance of the login program for each terminal.

Login runs user shell in a child process after user authenticates.

User shell runs user commands as child processes.
Environment variables and property lists

- The environment variable array is a **property list**.
  - The **property list** construct is very common and useful!
  - Also commonly used for configuration files.
  - It goes by various names: Java plist, Windows Registry, INI files
- Each element of the list is a string: \texttt{“NAME=VALUE”}.
- The standard library has primitives to look up the VALUE corresponding to a NAME.
- In Unix systems: standard environment variables are handed down through the shell: they give programs lots of information about the environment.
- The parent specifies them to the \texttt{exec*} syscall.
#include <stdio.h>

int
main(int argc, char* argv[]) {
  int i;

  printf("arguments: %d\n", argc);
  for (i=0; i<argc; i++) {
    printf("%d: %s\n", i, argv[i]);
  }
}

chase$ cc –o prog1 prog1.c
chase$ ./forkexec prog1
arguments: 1
0: prog1
child 19178 exited with status 0
chase$ ./forkexec prog1 one 2 3
arguments: 4
0: prog1
  1: one
  2: 2
  3: 3
Child 19181 exited with status 0.
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[], char* envp[])
{
    int i;
    int count = atoi(argv[1]);

    for (i=0; i < count; i++) {
        printf("env %d: %s\n", i, envp[i]);
    }
}
Environment variables (rough)

chase$ cc –o env0 env0.c
chase$ ./env0

Segmentation fault: 11

chase$ ./env0 12

env 0: TERM_PROGRAM=Apple_Terminal
env 1: TERM=xterm-256color
env 2: SHELL=/bin/bash
env 3: TMPDIR=/var/folders/td/ng76cpqn4zl1wrs57hldf1vm0000gn/T/
env 4: Apple_PubSub_Socket_Render=/tmp/launch-OtU5Bb/Render
env 5: TERM_PROGRAM_VERSION=309
env 6: OLDPWD=/Users/chase/c210-stuff
env 7: TERM_SESSION_ID=FFCE3A14-1D4B-4B08...
env 8: USER=chase
env 9: COMMAND_MODE=unix2003
env 10: SSH_AUTH_SOCK=/tmp/launch-W03wn2/Listeners
env 11: __CF_USER_TEXT_ENCODING=0x1F5:0:0

chase$
Environment variables (safer)

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char* argv[], char* envp[]) {
    int i;
    int count;

    if (argc < 2) {
        fprintf(stderr, "Usage: %s <count>\n", argv[0]);
        exit(1);
    }
    count = atoi(argv[1]);

    for (i=0; i < count; i++) {
        if (envp == 0) {
            printf("env %d: nothing!\n", i);
            exit(1);
        }
        else if (envp[i] == 0) {
            printf("env %d: null!\n", i);
            exit(1);
        } else
            printf("env %d: %s\n", i, envp[i]);
    }
}
```
Where do environment variables come from?

chase$ cc –o env env.c
chase$ ./env
chase$ ./forkexec env
Usage: env <count>
child 19195 exited with status 1
chase$ ./forkexec env 1
env 0: null!
child 19263 exited with status 1
chase$
forkexec revisited

char *lala = "lalala\n";
char *nothing = 0;
...
main(int argc, char *argv[]) {
    int status;
    int rc = fork();
    if (rc < 0) {
        ...
    } else if (rc == 0) {
        argv++;
        execve(argv[0], argv, &lala);
    } else {
        ...
    }
}

chase$ cc –o fel forkexec-lala.c
chase$ ./fel env 1
env 0: lalala
child 19276 exited with status 0
chase$
forkexec revisited again

... main(int argc, char *argv[], char *envp[])
  {
    int status;
    int rc = fork();
    if (rc < 0) {
      ...
    } else if (rc == 0) {
      argv++;
      execve(argv[0], argv, envp);
    } else {
      ...
    }
  }

chase$ cc -o fe forkexec1.c
chase$ ./fe env 3
env 0: TERM_PROGRAM=Apple_Terminal
env 1: TERM=xterm-256color
env 2: SHELL=/bin/bash
child 19290 exited with status 0
chase$
How about this?

chase$ ./fe fe fe fe fe fe fe fe fe fe fe fe fe env 3

<????>
How about this?

chase$ ./fe fe fe fe fe fe fe fe fe fe fe fe fe env 3
env 0: TERM_PROGRAM=Apple_Terminal
env 1: TERM=xterm-256color
env 2: SHELL=/bin/bash
child 19303 exited with status 0
child 19302 exited with status 0
child 19301 exited with status 0
child 19300 exited with status 0
child 19299 exited with status 0
child 19298 exited with status 0
child 19297 exited with status 0
child 19296 exited with status 0
child 19295 exited with status 0
child 19294 exited with status 0
child 19293 exited with status 0
child 19292 exited with status 0
chase$

It is easy for children to **inherit** environment variables from their parents.

Exec* enables the parent to control the environment variables and arguments passed to the children.

The child **process** passes the environment variables “to itself” but the parent **program** controls it.
Isolation

We need protection domains and protected contexts (“sandboxes”), even on single-user systems like your smartphone. There are various dimensions of isolation for protected contexts (e.g., processes):

- **Fault isolation.** One app or app instance (context or process) can fail independently of others. If it fails, the OS can kill the process and reclaim all of its memory, etc.

- **Performance isolation.** The OS manages resources (“metal and glass”: computing power, memory, disk space, I/O throughput capacity, network capacity, etc.). Each instance needs the “right amount” of resources to run properly. The OS prevents apps from impacting the performance of other apps. E.g., the OS can prevent a program with an endless loop from monopolizing the processor or “taking over the machine”. (How?)

- **Security.** An app may contain malware that tries to corrupt the system, steal data, or otherwise compromise the integrity of the system. The OS uses protected contexts and a reference monitor to check and authorize all accesses to data or objects outside of the context.