1.3 A Simple OS

- *OS Structure*
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- Files

A Simple OS

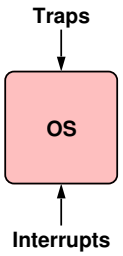
- Sixth-Edition Unix
  - source license available to universities in 1975 from Bell Labs
  - had major influence on modern OSes
    - Solaris
    - Linux
    - MacOS X
    - Windows
- Fits into 64KB of momory
  - single executable, completely stored in a *single file*
  - *loaded* into memory as the OS boots
  - *monolithic OS*

User vs. Privileged Modes

- *Processor modes:* part of the processor state (recall from your computer organization/architecture class regarding "processor")
  - most computers have at least two modes of execution
    - *user mode:* fewest privileges
    - *privileged mode:* most privileges
      - the only code that runs in this mode is part of the OS
- For Sixth-Edition Unix
  - the whole OS run in the privileged mode
  - everything else is an application and run in the user mode
- For other systems
  - major subsystems providing OS functionality may run in the user mode
- We use the word *"kernel"* to mean the portion of the OS that runs in privileged mode
  - sometimes, a subset of this

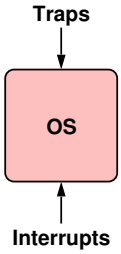
A Simple OS Structure

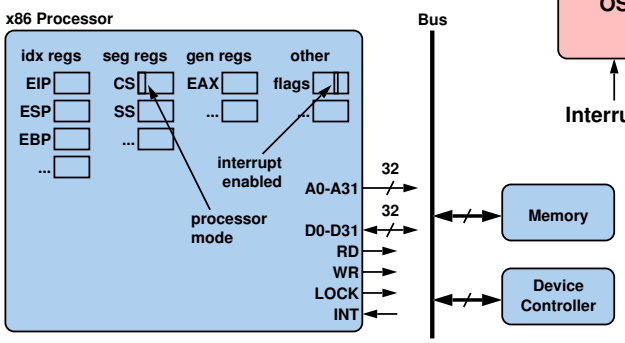
- Application programs call upon the OS via *traps*
- External devices call upon the OS via *interrupts*



A Simple OS Structure

- Application programs call upon the OS via *traps*
- External devices call upon the OS via *interrupts*





Traps

- *Traps* are the general means for invoking the kernel from user code
  - although we usually think of traps as *errors*
    - divide by zero, segmentation fault, bus error, etc.
  - but they don't have to be
    - *system calls*, *page fault*, etc.
- Traps always elicit some sort of response
  - for programming errors, the default action is to *terminate* the user program
  - for system calls, the OS is asked to perform some service

A Special Kind Of Trap - System Calls

- Invoking OS functionality in the kernel is more complex
  - but we want to make it look simple to applications
  - must be done carefully and correctly
    - really cannot trust the application programmers to do the right thing every time
- Provide *system calls* through which user code can access the kernel *in a controlled manner*
  - any necessary checking on whether the request should be permitted can be done in the system call
    - all done in user mode
  - if all goes well
    - sets things up
    - *traps* into the kernel by executing a special machine instruction, i.e., the "trap" machine instruction
    - the kernel figures out why it was invoked and handles the trap

Interrupts

- An *interrupt* is a request from an *external device* for a response from the *processor*
  - handled independently of any user program
    - unlike a trap, which is handled as part of the program that caused the trap
      - response to a trap directly affects that program
  - response to an interrupt may or may not indirectly affect the currently running program
    - often has *no direct effect* on the currently running program
- There's also something called *software interrupt*
  - generated programmatically (i.e., not by a device) by executing an "interrupt" machine instruction
  - this is very different from a hardware interrupt, although the mechanisms of handling interrupts are all very similar

Upcall

- A program may establish a handler *(i.e., a signal handler)* to be invoked in response to the error
  - the handler might clean up after the error and then terminate the program, or it might perform corrective action and continue with normal execution
- The *upcall* mechanism
  - *signals* allow the kernel to invoke code that's part of user program
    - for example, you can set a timer to expire at a certain time, when it expires, the OS can use the upcall mechanism to call a specified user function

1.3 A Simple OS

- OS Structure
- *Processes, Address Spaces, & Threads*
- Managing Processes
- Loading Program Into Processes
- Files

Program Execution

- Fundamental *abstraction* of *program execution*
  - memory
    - address space
      - things that are addressable by the program are kept together here
    - in Sixth-Edition Unix, processes do not share address space
    - recall that *process* is an abstraction of *memory*
  - processor(s)
    - recall that *thread* is an abstraction of *processor*
  - "execution context"
    - the *state* of a process and its threads
    - exactly "where you are" in the program
    - a thread needs some sort of a context to execute
- Note: multiple meanings of the word "context" in this class
  - *save context* and *restore context*
  - *thread context* vs. *interrupt context*

Program Execution

- With abstraction, comes an interface / API
  - for processes
    - `fork()`, `exec()`, `wait()`, `exit()`

A Program

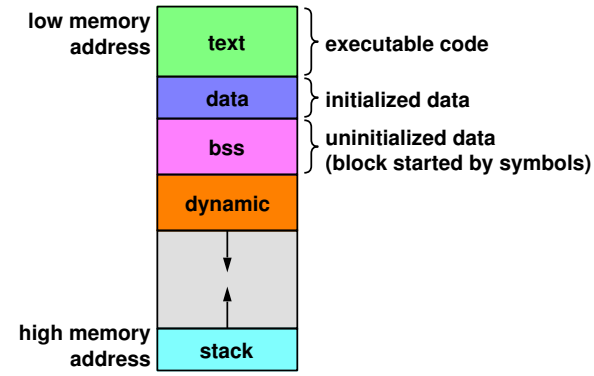
```
const int nprimes = 100;
int prime[nprimes];
int main() {
  int i;
  int current = 2;
  prime[0] = current;
  for (i=1; i<nprimes; i++) {
     int j;
  NewCandidate:
     current++;
     for (j=0; prime[j]*prime[j] <= current; j++) {
        if (current % prime[j] == 0)
           goto NewCandidate;
     }
     prime[i] = current;
  }
  return(0);
}
```

- My color codes for code
  - reserved words at in blue
  - numeric and string constants are in red
  - comments in green
  - black otherwise

Turing Machine Model of Computation

- A *Turing Machine* consists of
  - an infinite tape which is divided into cells, one next to the other *(i.e., infinite storage)*
    - one symbol in each cell (or can be a blank symbol)
  - a head that can read and write symbols on the tape and move the tape left and right one (and only one) cell at a time
  - a state register that stores the state of the Turing machine, one of finitely many *(i,e., finite state)*
  - a *finite* table of instructions that, given the state the machine is currently in and the symbol it is reading on the tape tells the machine to do the following in sequence
    - either erase or write a symbol
    - move the head
    - assume the same or a new state as prescribed

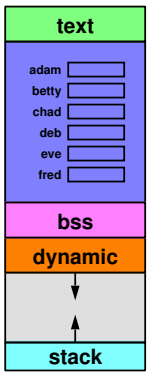
The Unix Address Space



- This is part of the *tape* of the Turing Machine
  - the rest of the *tape* of the Turing Machine can be reached by using the "*extended address space*"

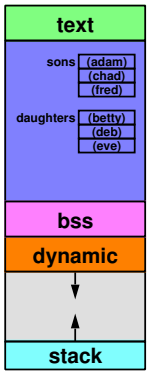
Note About Naming Objects

- How do you name objects in an address space
  - *"objects"* is the word we use to mean *any* data types (primitive, data structures, pointers)
- Variables
  - name each object



Note About Naming Objects

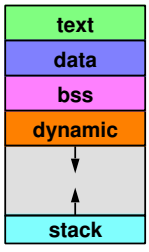
- How do you name objects in an address space
  - *"objects"* is the word we use to mean *any* data types (primitive, data structures, pointers)
- Variables
  - name each object
- Arrays
  - name an object with a *base* and an *index*
- Dynamically create objects do not have names
  - need *pointers*



Modified Program

```
int nprimes;              // in bss region
int *prime;               // in bss region
int main(int argc, char *argv[]) { // in stack
  int i;                  // in stack
  int current = 2;        // in stack
  nprimes = atoi(argv[1]);
  prime = (int*)malloc(nprimes*sizeof(int));
  prime[0] = current;
  for (i=1; i<nprimes; i++) {
     ...
  }
  return(0);
}
```

- where do all the variables reside?
- what is `argv[1]` and why `atoi()`?
- what is `sizeof()`?
- what does `malloc()` do?



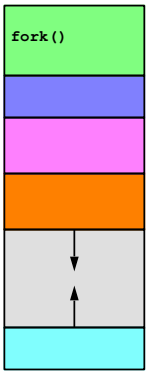
1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- *Managing Processes*
- Loading Program Into Processes
- Files

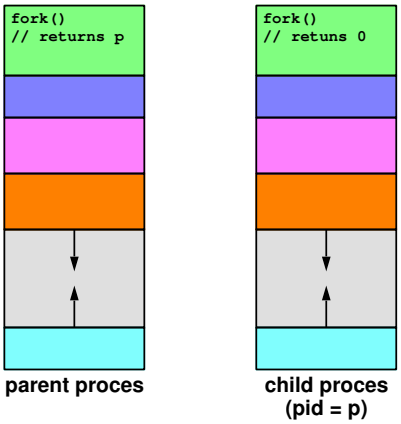
Creating a Process

- Creating a process is deceptively simple
  - make a copy of a process (the parent process)
    - `pid_t fork(void)`
    - the process where `fork()` is called is the *parent* process
    - the copy is the *child* process
    - in a way, `fork()` returns twice
      - once in the parent, the returned value is the *process ID (PID)* of the child process
      - once in the child, the returned value is 0
      - a PID is 16-bit long
  - this is the *only* way to create a process
- Making a copy of the entire address space can be expensive
  - Ch 7 shows speed up tricks
  - e.g., text segment is read-only so parent and child can share it
- Example: relationship between a shell (i.e., a command interpreter, such as `/bin/tcsh`) and `/bin/ls`

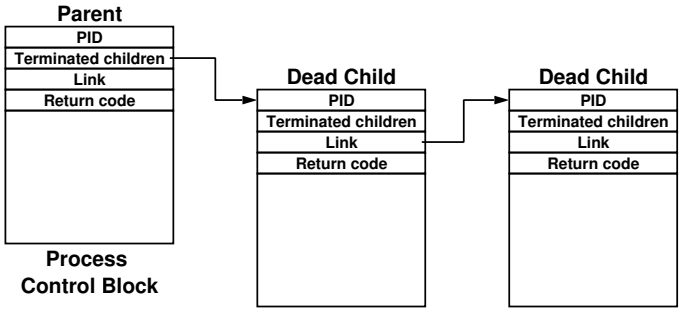
Creating a Process: Before



Creating a Process: After



Process Control Blocks



- *Process Control Block (PCB)* is a kernel data structure
  - pretty much every field is unsigned
  - return code (when a process dies) is 8-bit long
    - so that the parent process can know what happened to child
  - the "Link" field points to the next PCB
    - but, the next PCB in what list?

Fork and Wait

```
short pid;
if ((pid = fork()) == 0) {
  /* some code is here for the child to execute */
  exit(n);
} else {
  int ReturnCode;
  while(pid != wait(&ReturnCode))
    ;
  /* the child has terminated with ReturnCode as
     its return code */
}
```

- e.g., `/bin/tcsh` forks `/bin/ls`
- what does `exit(n)` do other than copying `n` into PCB?
  - least significant 8-bits of `n`
- what happens when `main()` calls `return(n)`?
  - eventually, `exit(n)` will be invoked
- `pid_t wait(int *status)` is a blocking call
  - it reaps dead child processes one at a time

Process Termination Issues

- PID is only 16-bits long
  - OS must not reuse PID too quickly or there may be ambiguity
- When `exit()` is called, the OS must not free up PCB too quickly
  - parent needs to get the return code
  - it's okay to free up everything else (such as address space)
- Solutions for both is for the terminated child process to go into a *zombie* state
  - only after `wait()` returned with the child's PID and the PID be reused and the PCB be freed up
  - but what if the parent calls `exit()` while the child is in the zombie state?
    - process 1 (the process with PID=1) inherits all the zombie children of this parent process
    - process 1 keeps calling `wait()` to reap the zombies

1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- *Loading Program Into Processes*
- Files

Loading Programs Into Processes

- How do you run a program?
  - make a copy of a process
    - any process
  - replace the child process with a new one
    - wipe out the child process
      - not everything, some stuff survives this (i.e., won't get destroyed)
    - using a family of system calls known as *exec*
  - kind of a waste to make a copy in the first place
    - but it's the only way
    - also, the OS does not know if the reason the parent process calls `fork()` is to run a new program or not

Exec

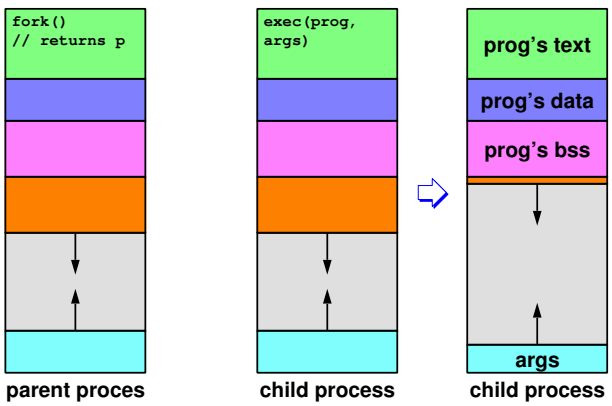
```
int pid;
if ((pid = fork()) == 0) {
  /* we'll discuss what might take place before
     exec is called */
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- what does `execl()` do?
  - "man execl" says:

```
int execl(const char *path,
          const char *arg, ...);
```

  - isn't "primes" in the 2nd argument kind of redundant?
  - what's up with "..."?
    - this is called *"varargs"* (similar to `printf()`)

Loading a New Image



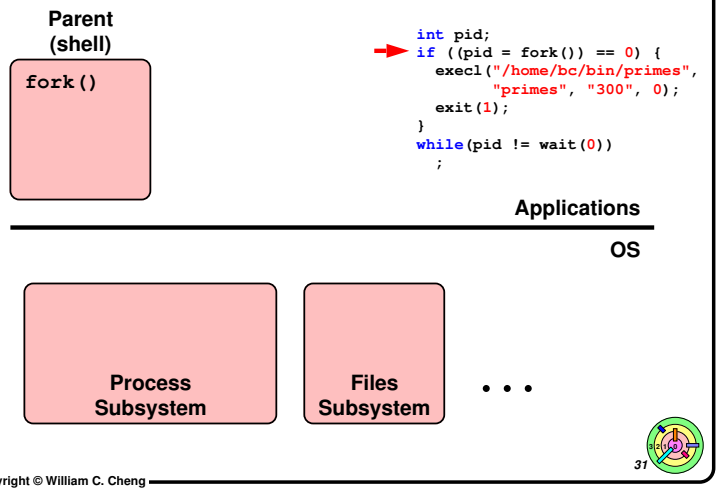
Exec

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
while(pid != wait(0)) /* ignore the return code */
  ;
```

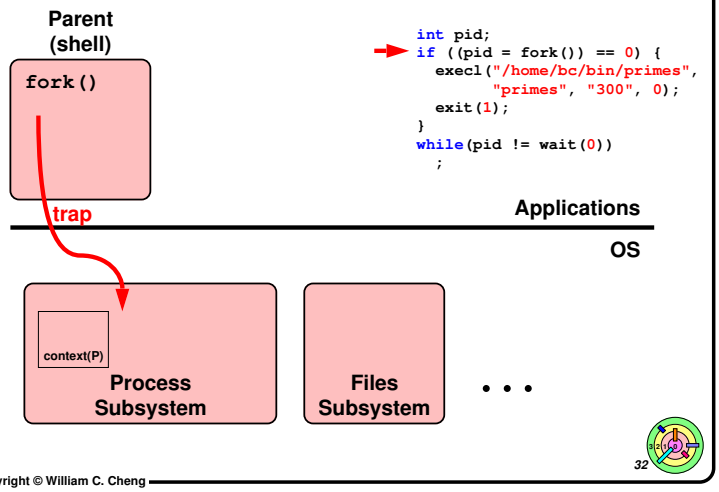
```
% primes 300
```

- Your login shell forks off a child process, load the primes program on top of it, wait for the child to terminate
  - the same code as before
  - `exit(1)` would get called if somehow `execl()` returned
    - if `execl()` is successful, it cannot return since the code is gone (i.e., the code segment has been replaced by the code segment of "primes")

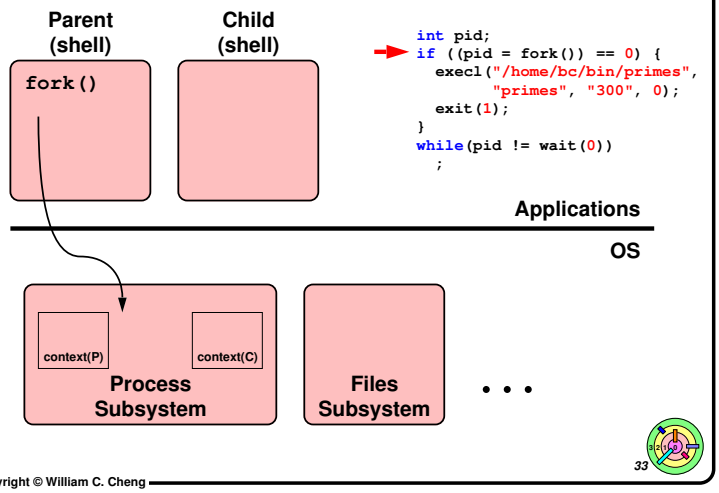
Put It All Together



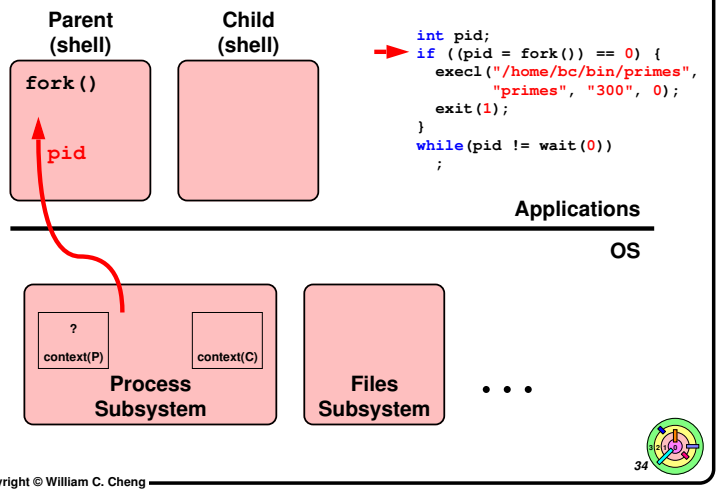
Put It All Together



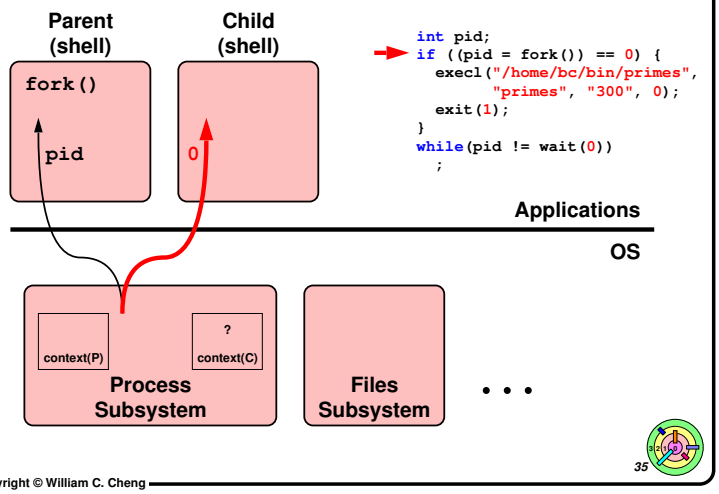
Put It All Together



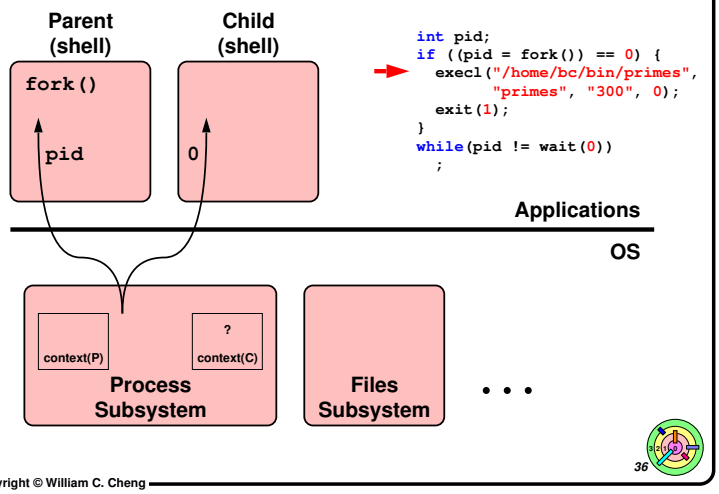
Put It All Together



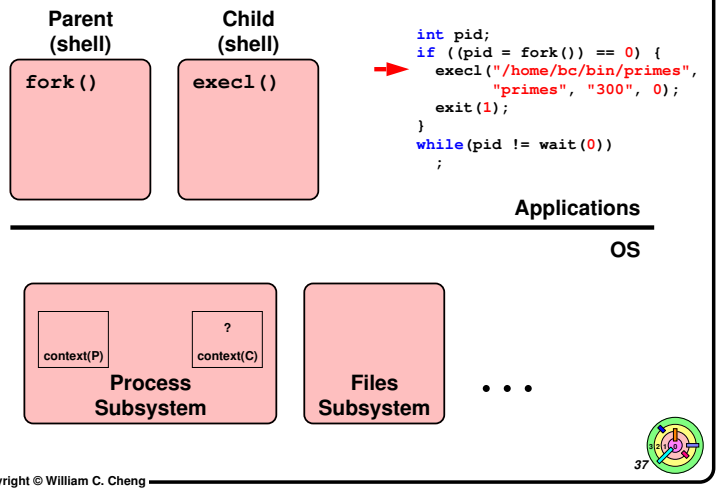
Put It All Together



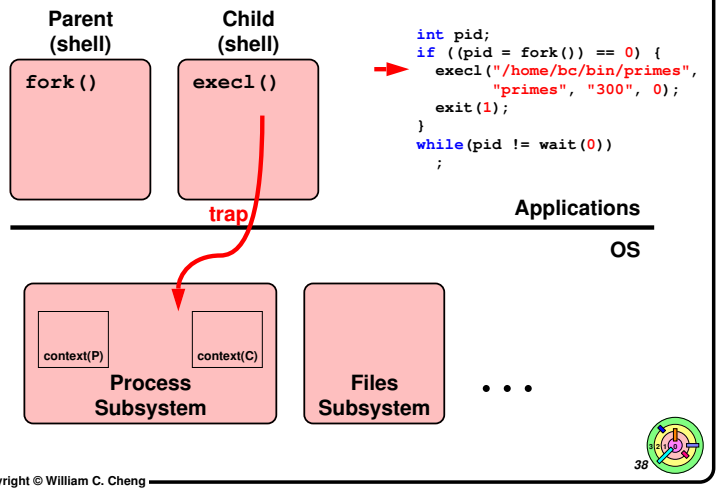
Put It All Together



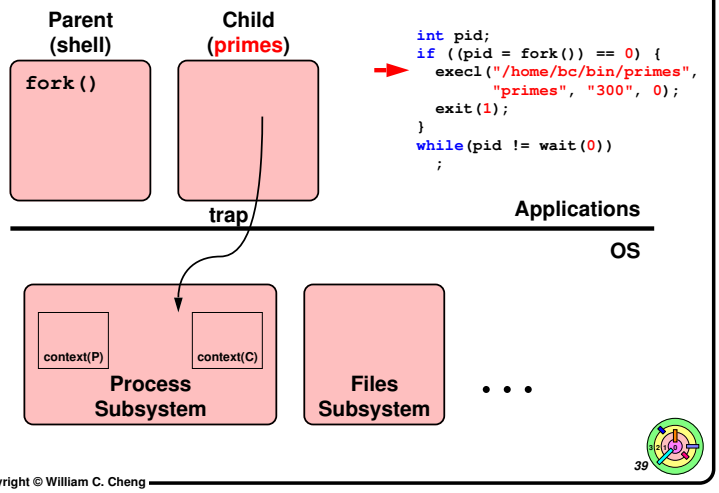
Put It All Together



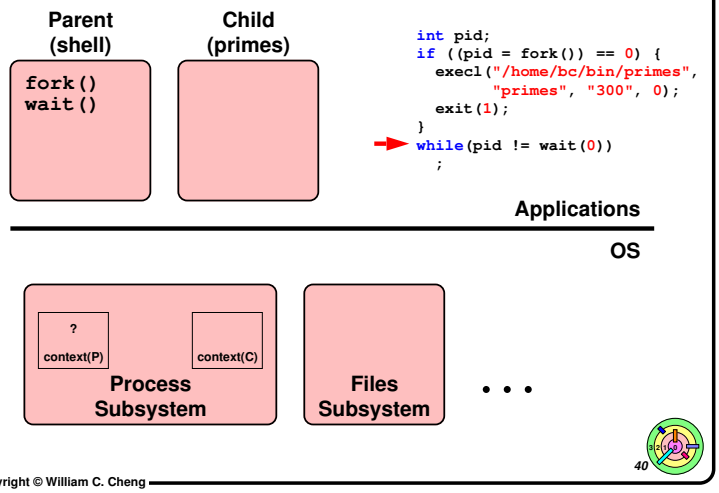
Put It All Together



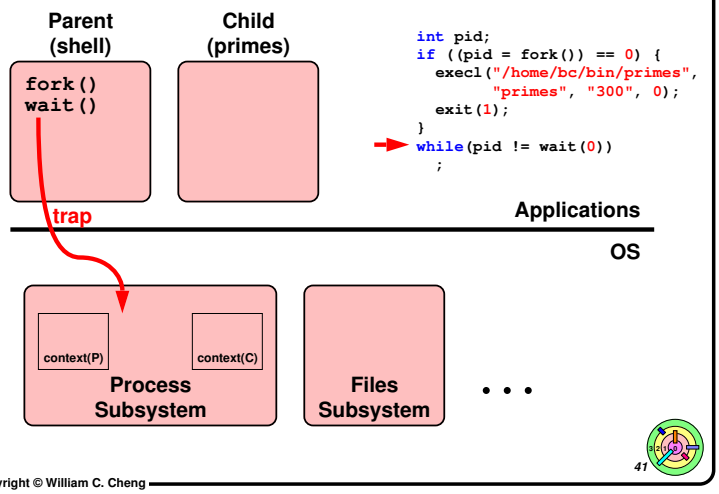
Put It All Together



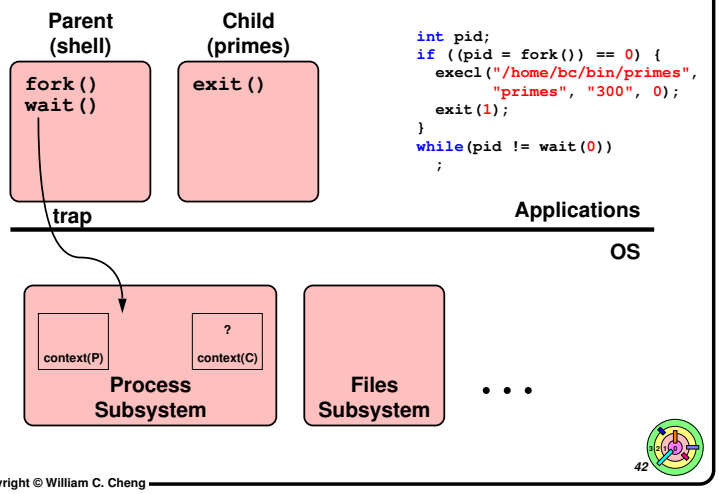
Put It All Together



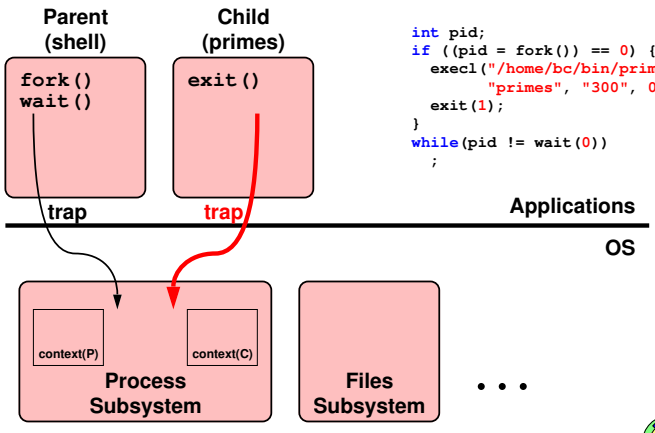
Put It All Together



Put It All Together

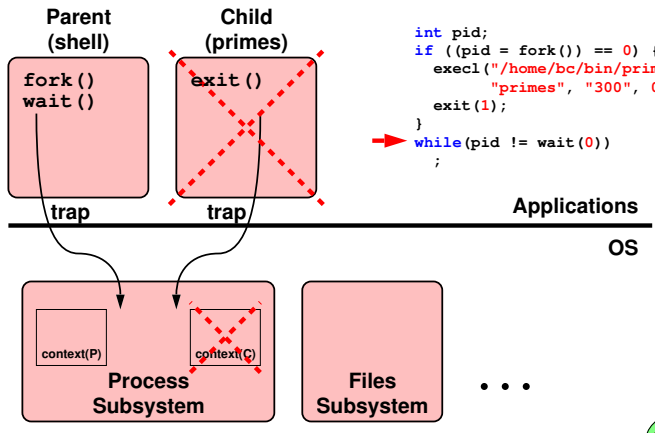


Put It All Together



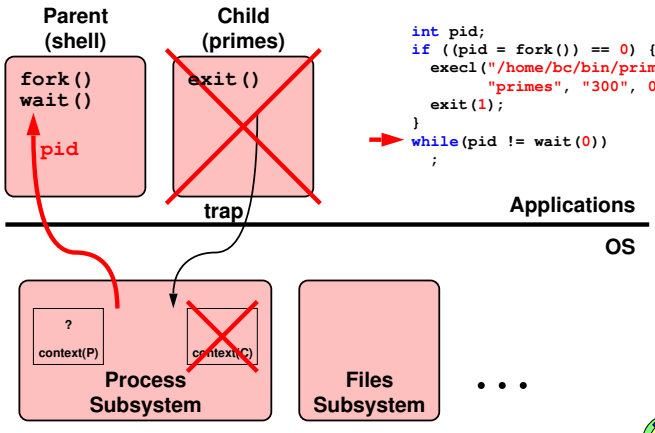

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```


Put It All Together



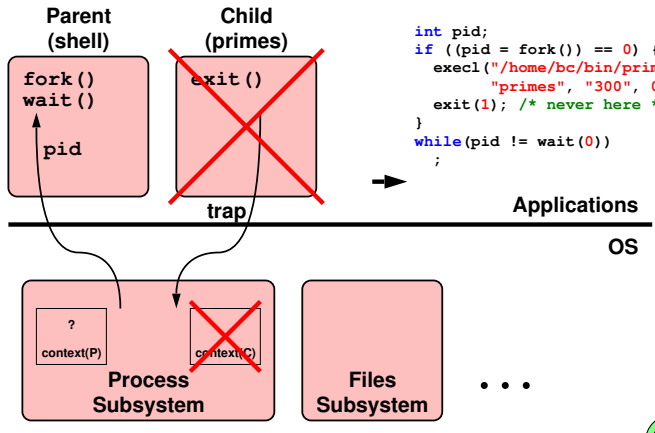

```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```


Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```


Put It All Together




```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1); /* never here */
}
while(pid != wait(0))
  ;
```

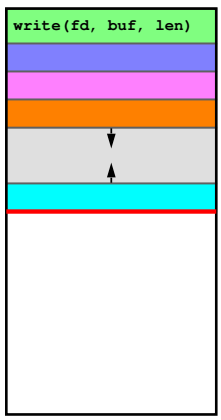

More On System Calls

- ***Sole interface*** between user and kernel
- Implemented as library routines that execute ***"trap" machine instructions*** to enter kernel
- Errors indicated by returning an invalid value
  - error code is in ***errno***

```
if (write(fd, buffer, bufsize) == -1) {
  // error!
  printf("error %d\n", errno);
  // see perror
}
```

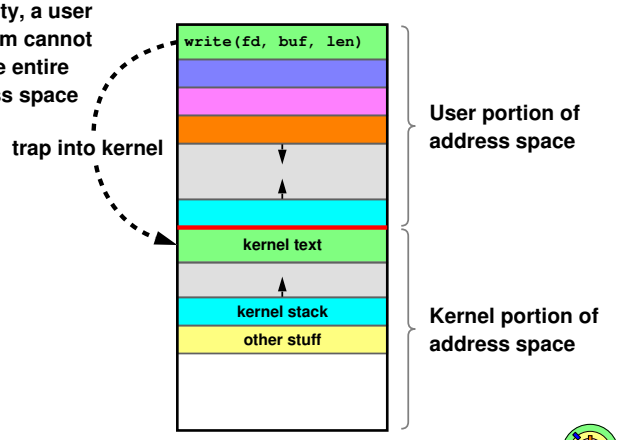

System Calls

- In reality, a user program cannot use the entire address space



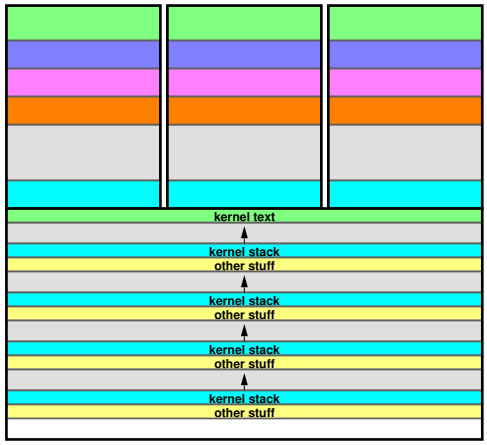
System Calls

- In reality, a user program cannot use the entire address space



- Is this the same *"thread of execution"*?

Multiple Processes



- the same kernel spans across all user processes
  - although there are other kernel processes as well (but they don't make system calls)

1.3 A Simple OS

- OS Structure
- Processes, Address Spaces, & Threads
- Managing Processes
- Loading Program Into Processes
- *Files*

Files

- Our "primes" program wasn't too interesting
  - it has no output!
  - cannot even verify that it's doing the right thing
  - other program cannot use its result
  - how does a process write to someplace *outside the process*?
- The notion of a *file* is our Unix system's *sole abstraction* for this concept of "someplace outside the process"
  - modern Unix systems have additional abstractions
- Files
  - abstraction of persistent data storage
  - means for fetching and storing data outside a process
    - including disks, another process, keyboard, display, etc.
    - need to *name* these different places
      - hierarchical naming structure
    - part of a process's *extended address space*
      - file "cursor position" is part of "execution context"

Naming Files

- Directory system
  - shared by all processes running on a computer
    - although each process can have a different view
    - Unix provides a means to restrict a process to a subtree
      - by redefining what "root" means for the process
  - name space is outside the processes
    - a user process provides the name of a file to the OS
    - the OS returns a *handle* to be used to access the file
      - after it has verified that the process is allowed *access* along the *entire path*, starting from root
    - user process uses the handle to read/write the file
      - avoid subsequent access checks
- *Using a handle* (which can be an index into a kernel array) to *refer to an object managed by the kernel* is an important concept
  - *handles* are essentially an *extension* to the process's *address space*
    - can even *survive execs*!

The File Abstraction

- A file is a simple array of bytes
- Files are made larger by writing beyond their current end
- Files are named by paths in a naming tree
- System calls on files are *synchronous*
  - i.e., will not return until the operation is considered completed
- File API
  - `open()`, `read()`, `write()`, `close()`
  - e.g., `cat`

File Handles (File Descriptors)

```
int fd;
char buffer[1024];
int count;
if ((fd = open("/home/bc/file", O_RDWR) == -1) {
  // the file couldn't be opened
  perror("/home/bc/file");
  exit(1);
}
if ((count = read(fd, buffer, 1024)) == -1) {
  // the read failed
  perror("read");
  exit(1);
}
// buffer now contains count bytes read from the file
```

- what is O_RDWR?
- what does `perror()` do?
- *cursor* position in an opened file depends on what functions/system calls you use
  - what about C++?

Standard File Descriptors

- Standard File Descriptors
  - 0 is `stdin` (by default, "map/connect" to the keyboard)
  - 1 is `stdout` (by default, "map/connect" to the display)
  - 2 is `stderr` (by default, "map/connect" to the display)

```
main() {
  char buf[BUFSIZE];
  int n;
  const char *note = "Write failed\n";

  while ((n = read(0, buf, sizeof(buf))) > 0)
    if (write(1, buf, n) != n) {
      (void)write(2, note, strlen(note));
      exit(EXIT_FAILURE);
    }
  return(EXIT_SUCCESS);
}
```

Back to Primes

- Have our primes program write out the solution, i.e., the `primes[]` array

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
  ...
  for (i=1; i<nprimes; i++) {
    ...
  }
  if (write(1, prime, nprimes*sizeof(int)) == -1) {
    perror("primes output");
    exit(1);
  }
  return(0);
}
```

- the output is not readable by human

Human-Readable Output

```
int nprimes;
int *prime;
int main(int argc, char *argv[]) {
  ...
  for (i=1; i<nprimes; i++) {
    ...
  }
  for (i=0; i<nprimes; i++) {
    printf("%d\n", prime[i]);
  }
  return(0);
}
```

- please see the *Programming FAQ* regarding the difference between a *file descriptor* and a *file pointer*

Allocation of File Descriptors

- Whenever a process requests a new file descriptor, the lowest numbered file descriptor not already associated with an open file is selected; thus

```
#include <fcntl.h>
#include <unistd.h>
...
close(0);
fd = open("file", O_RDONLY);
```

- will always associate "file" with file descriptor 0 (assuming that `open()` succeeds)

- You will need to implement the above rule in the kernel 2 assignment

Running It

```
if (fork() == 0) {
  /* set up file descriptor 1 in the child process */
  close(1);
  if (open("/home/bc/Output", O_WRONLY) == -1) {
    perror("/home/bc/Output");
    exit(1);
  }
  execl("/home/bc/bin/primes", "primes", "300", 0);
  exit(1);
}
/* parent continues here */
while(pid != wait(0)) /* ignore the return code */
  ;
```

- `close(1)` removes file descriptor 1 from *extended address space*
- file descriptors are allocated *lowest first* on `open()`
- *extended address space* survives *execs*
- new code is same as running

```
% primes 300 > /home/bc/Output
```

I/O Redirection

```
% primes 300 > /home/bc/Output
```

- The ">" parameter in a shell command that instructs the command shell to *redirect* the output to the given file
  - If ">" weren't there, the output would go to the display
- Can also redirect input

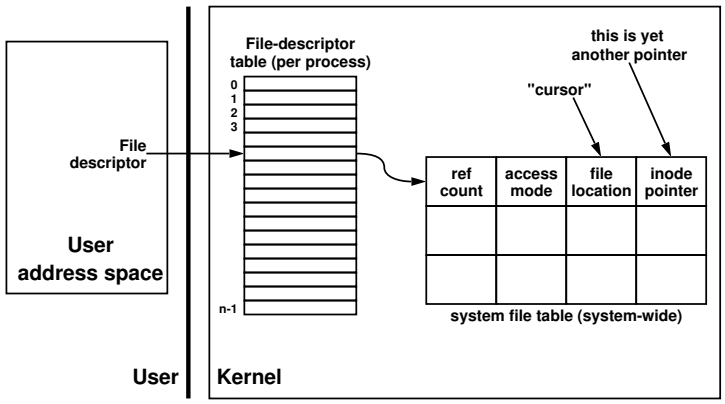
```
% cat < /home/bc/Output
```

  - when the "`cat`" program reads from file descriptor 0, it would get the data byes from the file "/home/bc/Output"

File-Descriptor Table

- A file descriptor refers not just to a file
  - it also refers to the *process's* current *context* for that file
    - includes how the file is to be accesses (how `open()` was invoked)
    - *cursor* position
- *Context* information must be maintained by the OS and not directly by the user program
  - let's say a user program opened a file with O_RDONLY
  - later on it calls `write()` using the opened file descriptor
  - how does the OS knows that it doesn't have write access?
    - stores O_RDONLY in context
  - if the user program can manipulate the context, it can change O_RDONLY to O_RDWR
  - therefore, user program must not have access to context!
    - all it can see is the handle
    - the file handle is an *index* into an array maintained for the process in kernel's address space

File-Descriptor Table



- context is not stored directly into the file-descriptor table
  - one-level of *indirection*