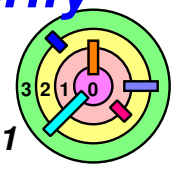
Housekeeping (Lecture 4 - 6/3/2025)

- **PA1 is due at 11:45pm on Tuesday, 6/3/2025**
  - **if you have code from current or a previous semester, *do not look at/copy/share* any code from it**
    - **it's best if you just get rid of it**
  - **if you include files that's not part of the original "`make pa1-submit`" command, the grader will delete them**

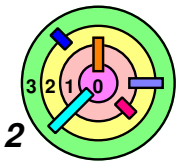
- ***Grading guidelines* is the *ONLY* way we will grade and we can only grade on a *standard 32-bit Ubuntu Linux 16.04 inside VirtualBox/UTM* or on AWS Free Tier**
  - **the *grading guidelines* is *part of the spec***
  - **although not recommended, you can do your development on a different platform**
    - **you must test your code on the "standard" platform because those are the *only* platforms the grader is allowed to grade on**
  - **if you make submission, make sure you run through the *Verify Your Submission* procedure as if you are the grader**



Ch 3: The Programming Interface

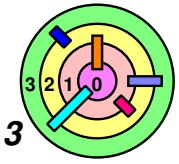
Bill Cheng

http://merlot.usc.edu/william/usc/



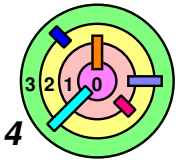
What Should OS Kernel Provide To Applications?

- What functions an operating system needs to provide applications?
  - *process management:* Can a program create an instance of another program? Wait for it to complete? Stop or resume another running program? Send it an asynchronous event?
  - *input/output:* How do processes communicate with devices attached to the computer and through them to the physical world? Can processes communicate with each other?
  - *thread management:* Can we create multiple activities or threads that share memory or other resources within a process? Can we stop and start threads? How do we synchronize their use of shared data structures?
  - *memory management:* Can a process ask for more (or less) memory space? Can it share the same physical memory region with other processes?
  - (cont...)

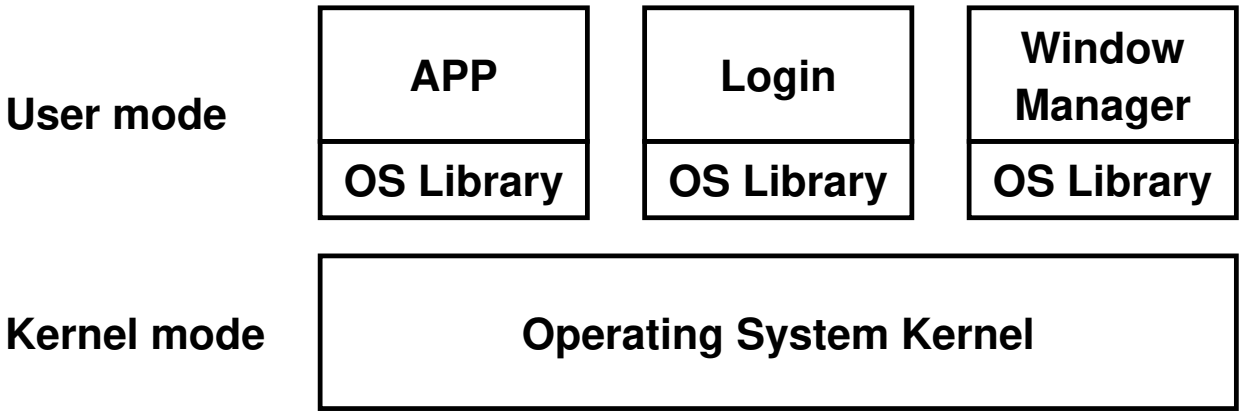


What Should OS Kernel Provide To Applications?

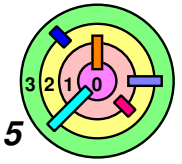
- What functions an operating system needs to provide applications?
  - *file systems and storage:* How does a process store the user's data persistently so that it can survive machine crashes and disk failures? How does the user name and organize their data?
  - *networking and distributed systems:* How do processes communicate with processes on other computers? How do processes on different computers coordinate their actions despite machine crashes and network problems?
  - *graphics and window management:* How does a process control pixels on its portion of the screen? How does a process make use of graphics accelerators?
  - *authentication and security:* What permissions does a user or a program have, and how are these permissions kept up to date? On what basis do we know the user (or program) is who they say they are?
- This chapter focuses on the first two



OS Functionalities

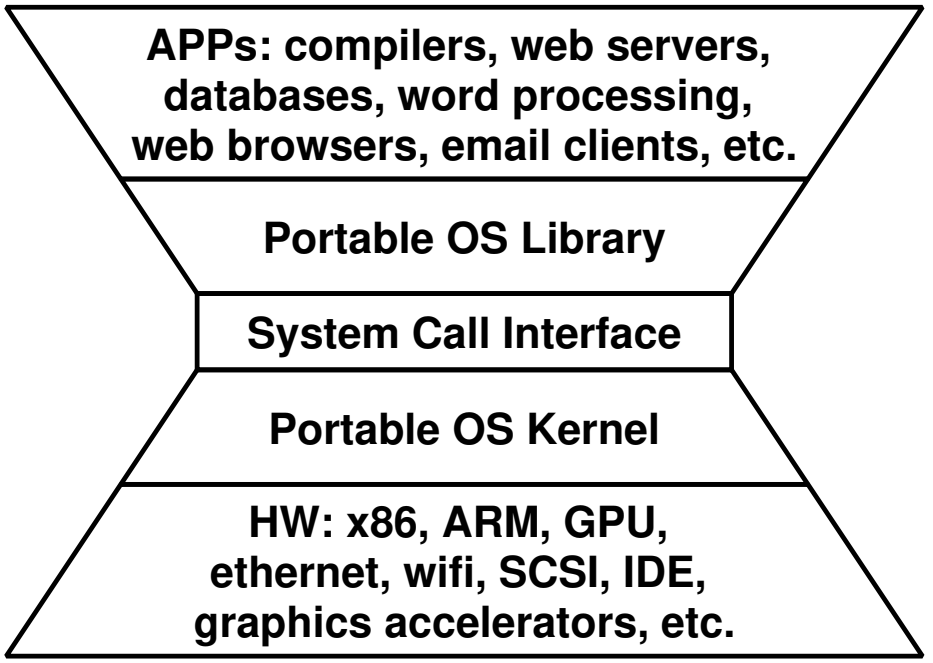


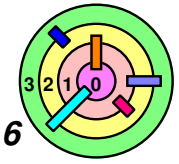
- OS functionalities can be implemented in *user-level programs*, in *user-level libraries*, in the *kernel* , or in a *user-level server* invoked by the kernel
- How do you choose where to implement an OS function then?
  - sometimes there is no clear winner
  - tradeoff between
    - safety - cannot be implemented in user-level library
    - reliability - monolithic kernel vs. microkernel design
    - performance - goes against safety and reliability
    - flexibility



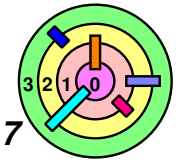
OS Functionalities

- **Flexibility:**
  - much easier to change code outside the kernel
  - try not to change the system call interface
  - UNIX design philosophy is to keep system call interface simple and powerful so that almost all innovation can happen in user code
  - the result is that UNIX system call interface is highly portable to new hardware without needing to rewrite application code



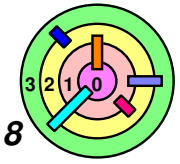


(3.1) Process Management



Process Management: Main Points

- Creating and managing processes
  - `fork()`, `exec()`, `wait()`
- Performing I/O
  - `open()`, `read()`, `write()`, `close()`
- Communicating between processes
  - `pipe()`, `dup()`, `select()`, `connect()`
- Example: implementing a shell



Shell

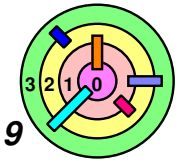
- A shell is a job control system
  - allows programmer to create and manage a set of programs to do some task
  - Windows, MacOS, Linux all have shells
- Example: to compile a C program

```
cc -c sourcefile1.c
cc -c sourcefile2.c
ln -o program sourcefile1.o sourcefile2.o
```

  - or:

```
gcc -c sourcefile1.c
gcc -c sourcefile2.c
gcc -o program sourcefile1.o sourcefile2.o
```

- Q: If the shell runs at user-level, what system calls does it make to run each of the programs
  - Ex: `cc`, `ln`



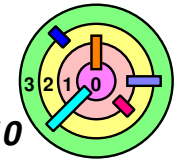
Shell Script

- You can put a bunch of commands into a file and have the shell execute from the file; this file is called a "shell script"
  - e.g., "`foo.sh`"

```
gcc -c sourcefile1.c
gcc -c sourcefile2.c
gcc -o program sourcefile1.o sourcefile2.o
```

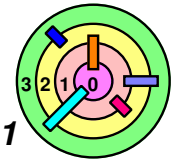
  - to execute all the commands in "`foo.sh`", do

```
source foo.sh
```



Windows `CreateProcess()`

- Windows has a `CreateProcess()` system call to create a new process to run a program
  - create and initialize the process control block (PCB) in the kernel
  - create and initialize a new *address space*
  - *load* the program into the address space
  - copy arguments into memory in the address space
  - initialize the hardware context to start execution at `start()`
  - inform the *scheduler* that the new process is ready to run
- Windows lets the parent process to control things like:
  - privileges of child process
  - where it sends its input and output
  - where to store its files
  - what to use as a scheduling priority
  - etc.

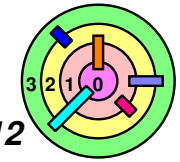


Windows `CreateProcess()` API (Simplified)

- End results for Windows:

```
if (!CreateProcess(
     NULL,     // No module name (use command line)
     argv[1], // Command line
     NULL,     // Process handle not inheritable
     NULL,     // Thread handle not inheritable
     FALSE,    // Set handle inheritance to FALSE
     0,        // No creation flags
     NULL,     // Use parent's environment block
     NULL,     // Use parent's starting directory
     &si,      // Pointer to STARTUPINFO structure
     &pi )     // Pointer to PROCESS_INFORMATION
               //     structure
)
```

- UNIX does this in two steps
  - `fork()` system call
  - `exec` system call

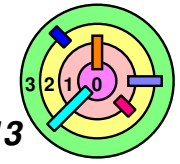


UNIX Process Management

- UNIX `fork()`: system call to create a complete copy of the current process, and runs it at the same place as the current process
  - `fork()` takes no arguments
  - when `fork()` returns, code written in the parent can set up privileges, priorities, and I/O to *set up context* for the child
- UNIX exec: a family of system calls to change the program being run by the current process
  - "`man exec`" on Linux shows:

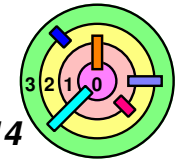
```
int execl(char *path, char *arg, ..., NULL);
int execlp(char *file, char *arg, ..., NULL);
int execle(char *path, char *arg, ..., NULL, char *envp[]);
int execv(char *path, char *argv[]);
int execvp(char *file, char *argv[]);
int execvpe(char *file, char *argv[], char *envp[]);
```

- UNIX `wait()`: system call to wait for a process to finish
- UNIX `signal()`: system call to send a *notification* to another process



The `fork()` System Call

- The kernel performs the following in the `fork()` system call implementation:
  - create and initialize the process control block (PCB) in the kernel
  - create a new address space
  - initialize the address space with a copy of the entire contents of the address space of the parent
  - inherit the execution context of the parent (e.g., any open files)
  - inform the scheduler that the new process is ready to run



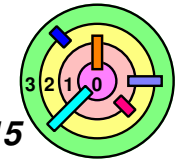
The `fork()` System Call

- Creating a process is deceptively simple
  - make a copy of a 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

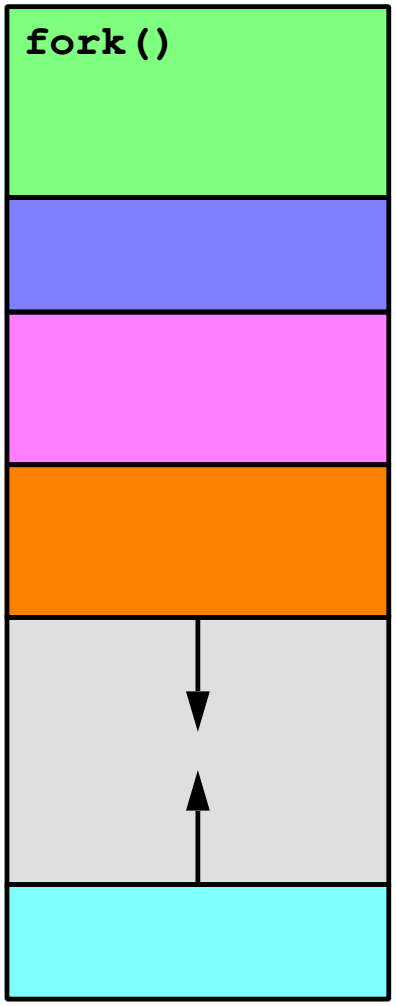
process control block

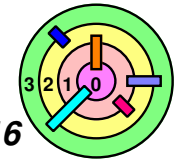
PID
... (other stuff) ...

- Ex: relationship between a commandline shell (e.g., `/bin/bash`) and `/bin/ls` (when you type "`ls`" and press `<ENTER>`)

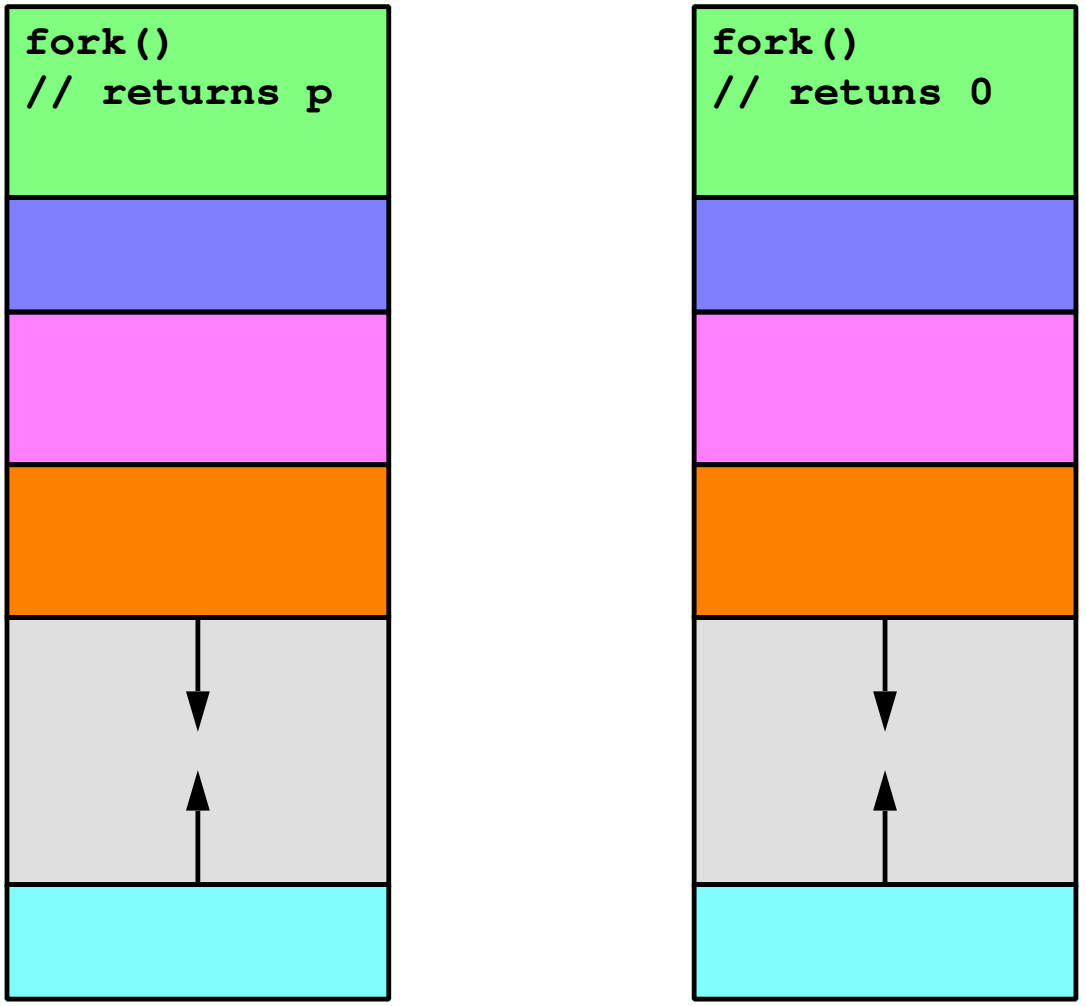


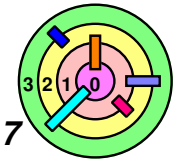
Creating a Process





Creating a Process





The `exit()` System Calls

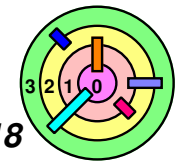
- The `exit()` system call

```
void exit(int status)
```

  - your process can call `exit(n)` to self-terminate
    - set `n` to be the "exit/return code" of this process
    - this sytem call does not return (your process will die inside the kernel)
  - exit/return code is 8 bits long
    - so that the parent process can know how the child process died

process control block

PID
exit status
... (other stuff) ...



The `wait()` System Calls

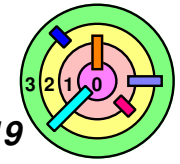
- The `wait()` system call

```
pid_t wait(int *status)
```

process control block

PID
exit status
... (other stuff) ...

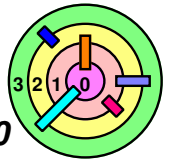
  - your process can call only `wait()` to wait for *any* child process to die (and cannot wait for a specific child process to die)
    - the `wait()` system call returns *two* values
      - C does not support pass-by-reference, so you have to pass the address of a variable (i.e., pass-by-pointer)
    - returns the PID of a dead child process and `(*status)` is the exit/return code of that child process
      - if there are more than one dead child processes, one of them will be *chosen at random*
    - if `status` is the NULL pointer, it means that the caller is not interested in the child exit/return code
  - `wait()` is a *blocking call*, i.e., the calling process may get suspended (in this case, inside the kernel) if this call cannot return yet (i.e., no dead child processes)



Fork and Wait

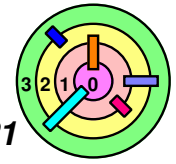
```
short pid;
if ((pid = fork()) == 0) {
   /* some code is here for the child to execute */
   exit(n);
} else {
   int ReturnCode;
   while(pid != wait(&ReturnCode))
      ;
   /* the child has terminated with ReturnCode as
      its return code */
}
```

- e.g., this is the first step when `/bin/bash` 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)`?
  - the startup function will call `exit(n)`
- `pid_t wait(int *status)` is a ***blocking call***
  - it reaps dead child processes ***one at a time***
- parent and child are the same "program" here!



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 can the PID be reused and the PCB can 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 children of this parent process
      - this is known as *"reparenting"*
    - process 1 keeps calling `wait()` to reap the zombies



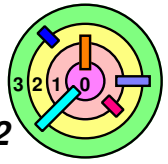
Exec

```
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 */
   ;
```

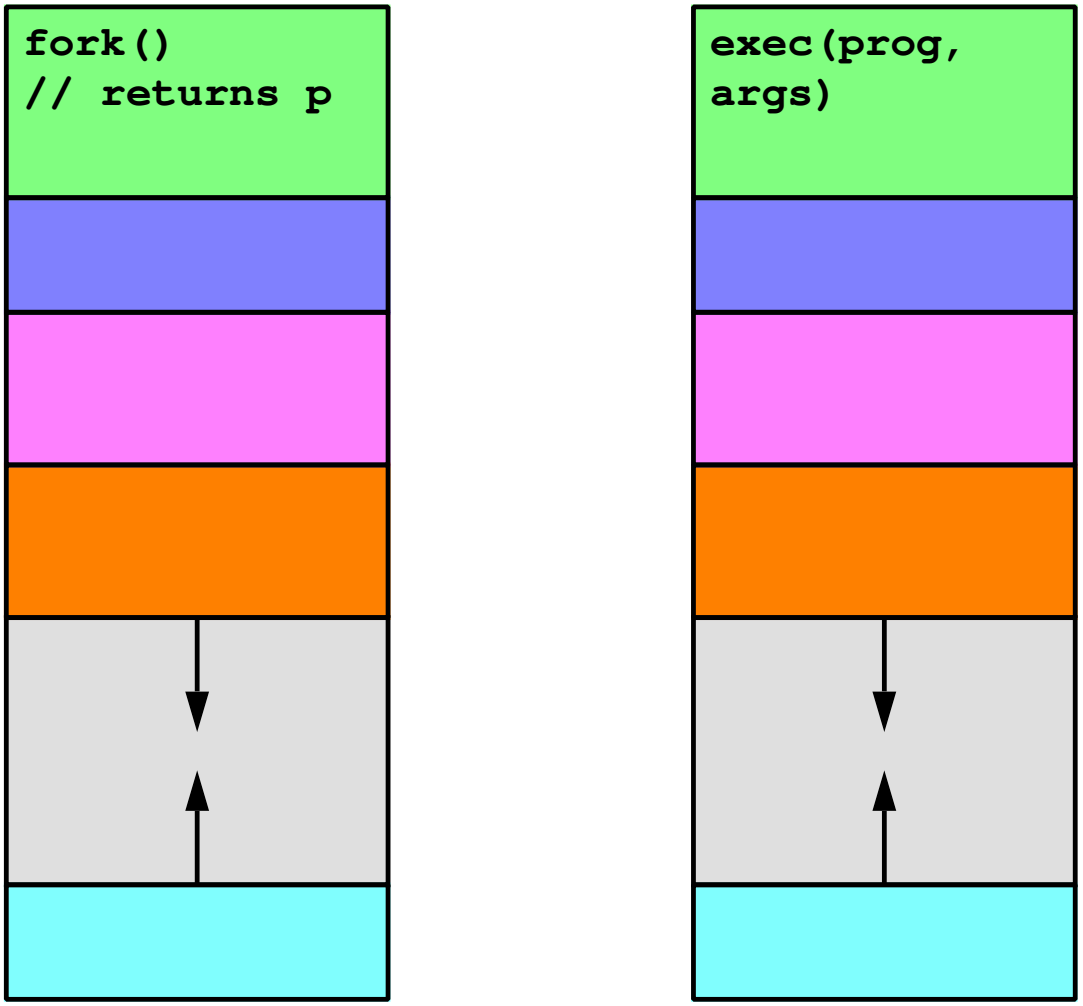
- what does `execl()` do?
  - "man execl" says:

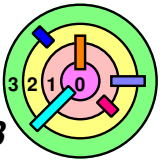
```
int execl(const char *path,
          const char *arg, ...);
```

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

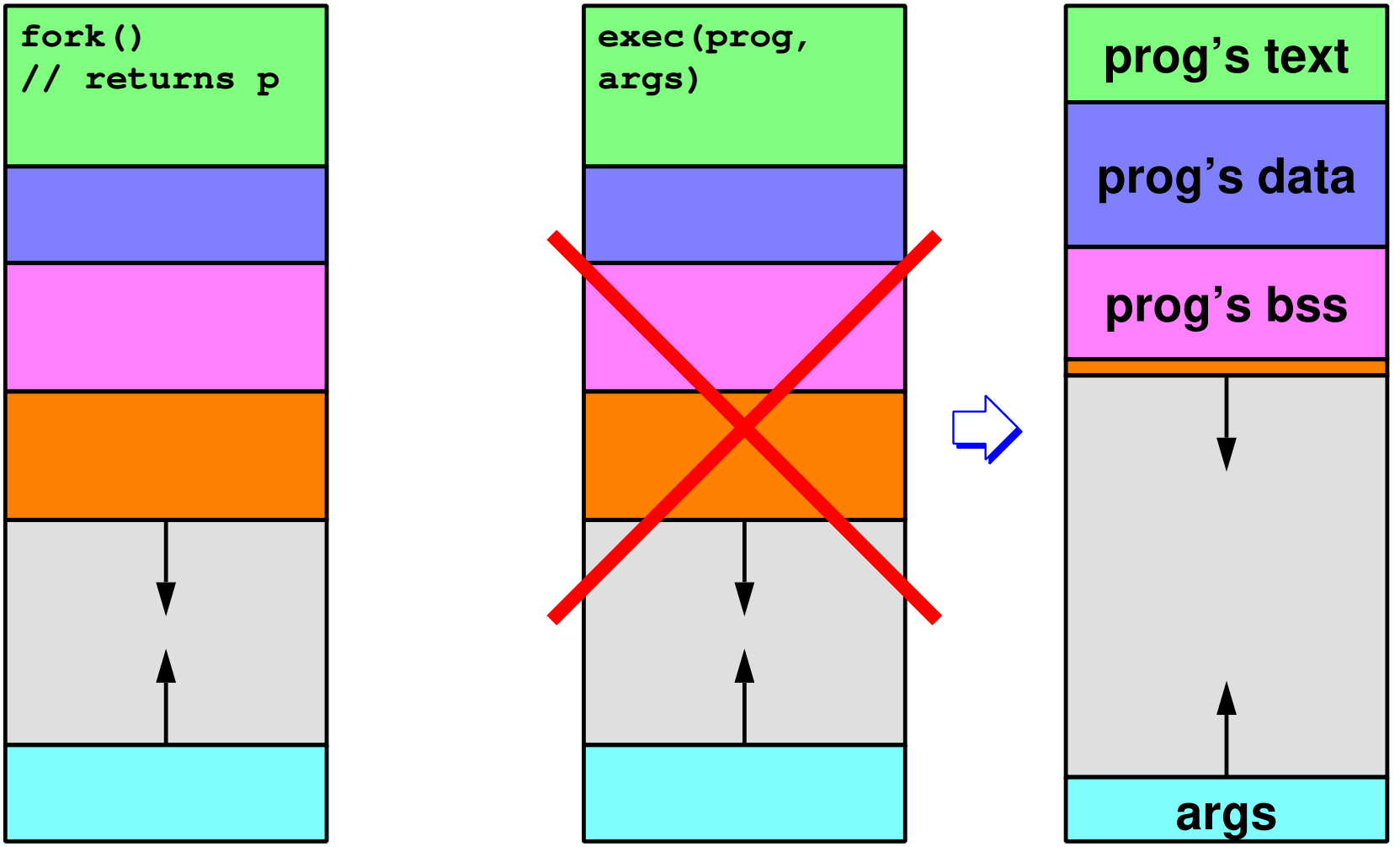


Loading a New Image





Loading a New Image

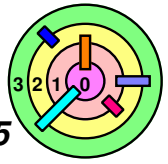


Exec

```
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

Parent (shell)

fork()

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

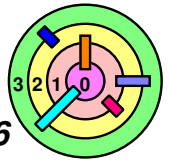
Applications

OS

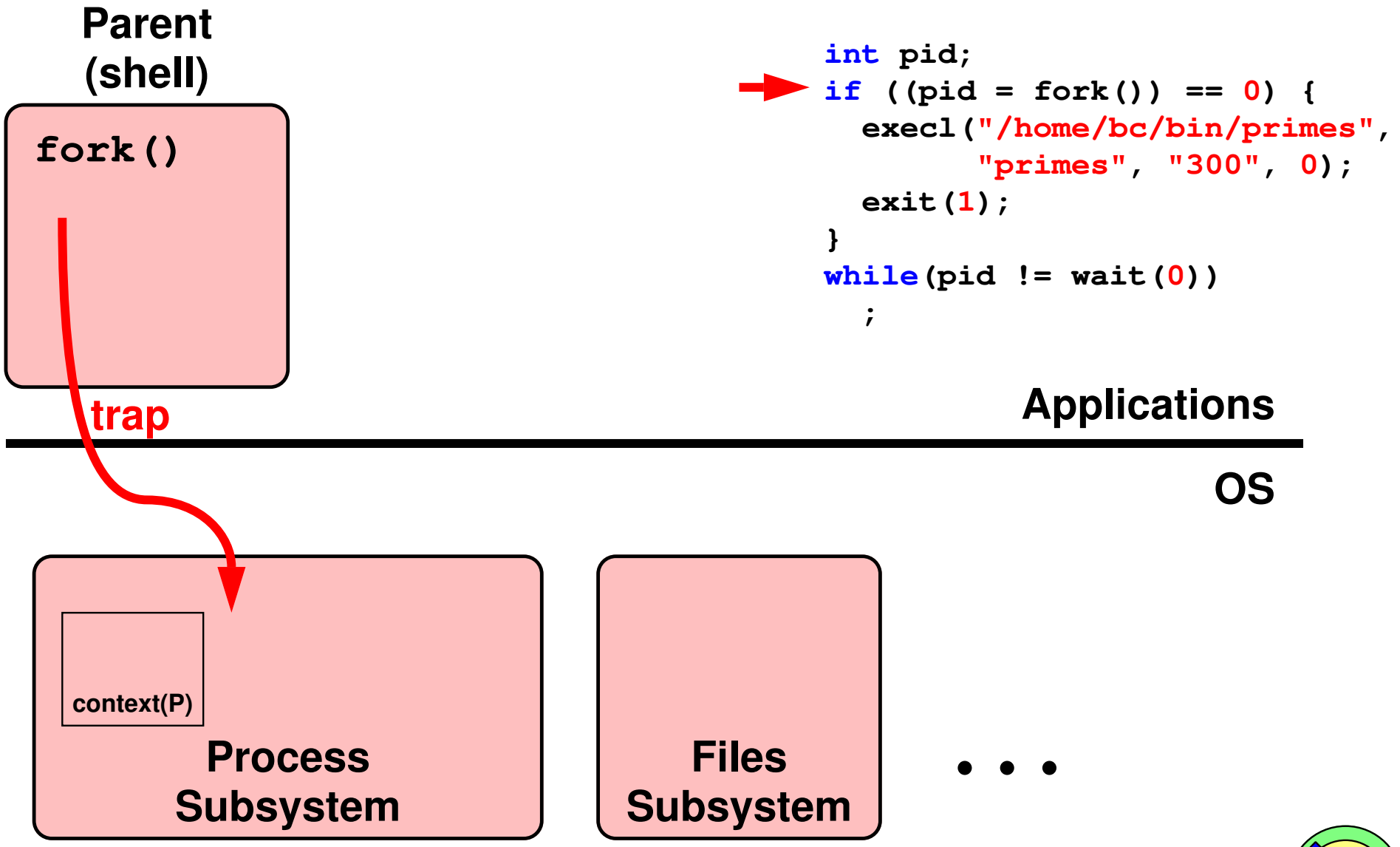
Process Subsystem

Files Subsystem

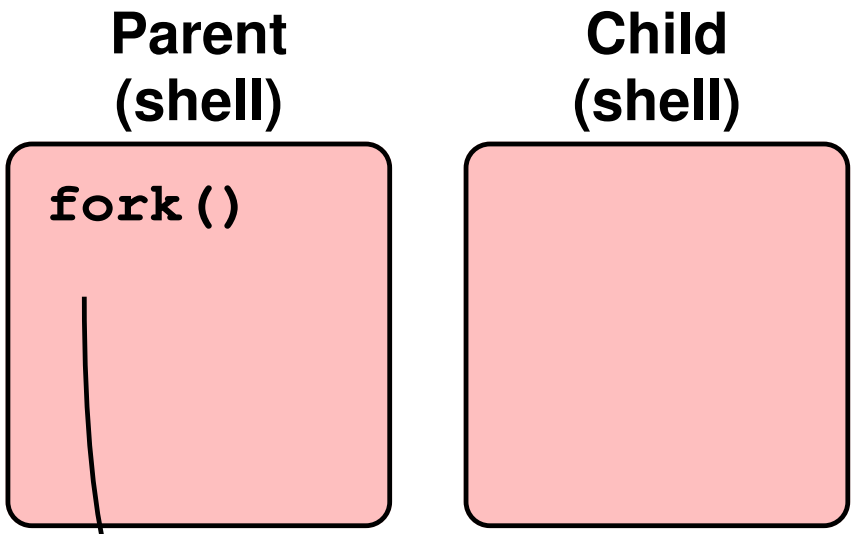
. . .



Put It All Together



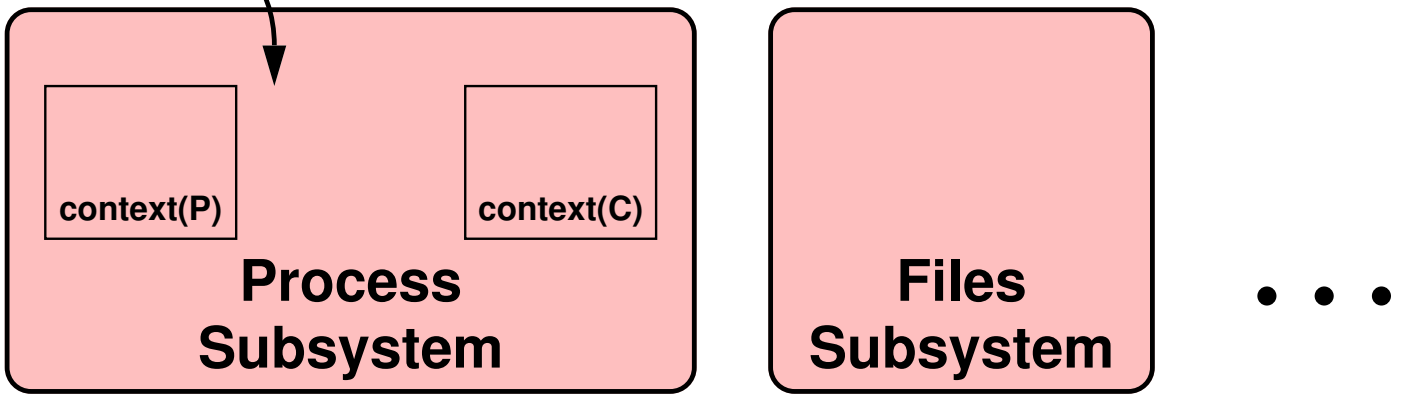
Put It All Together

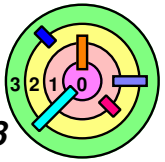



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

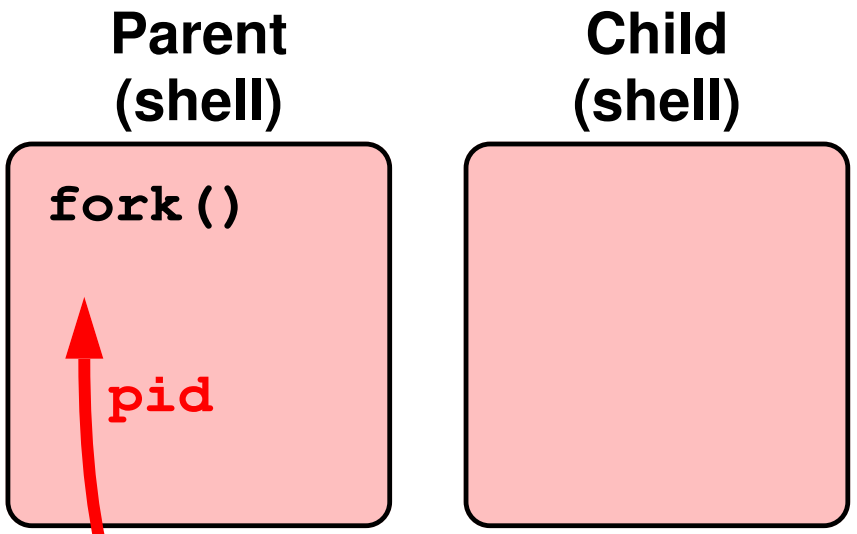
Applications

OS





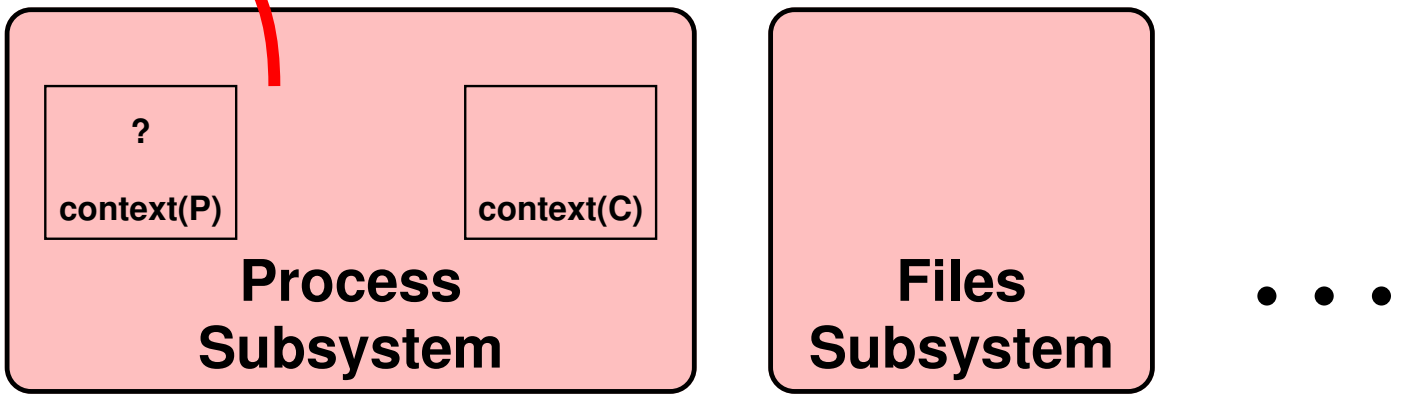
Put It All Together

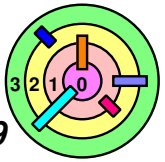



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

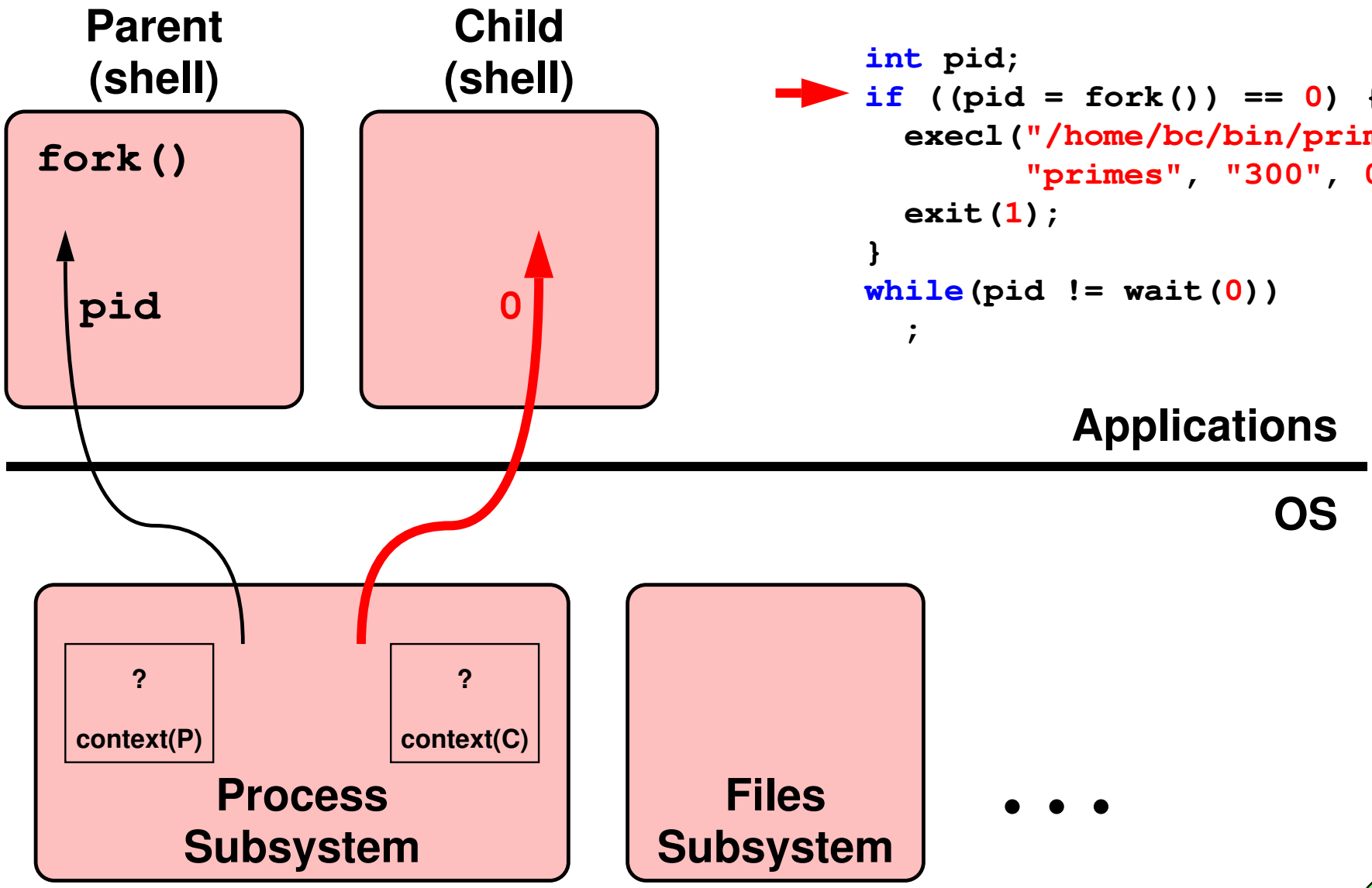
Applications

OS

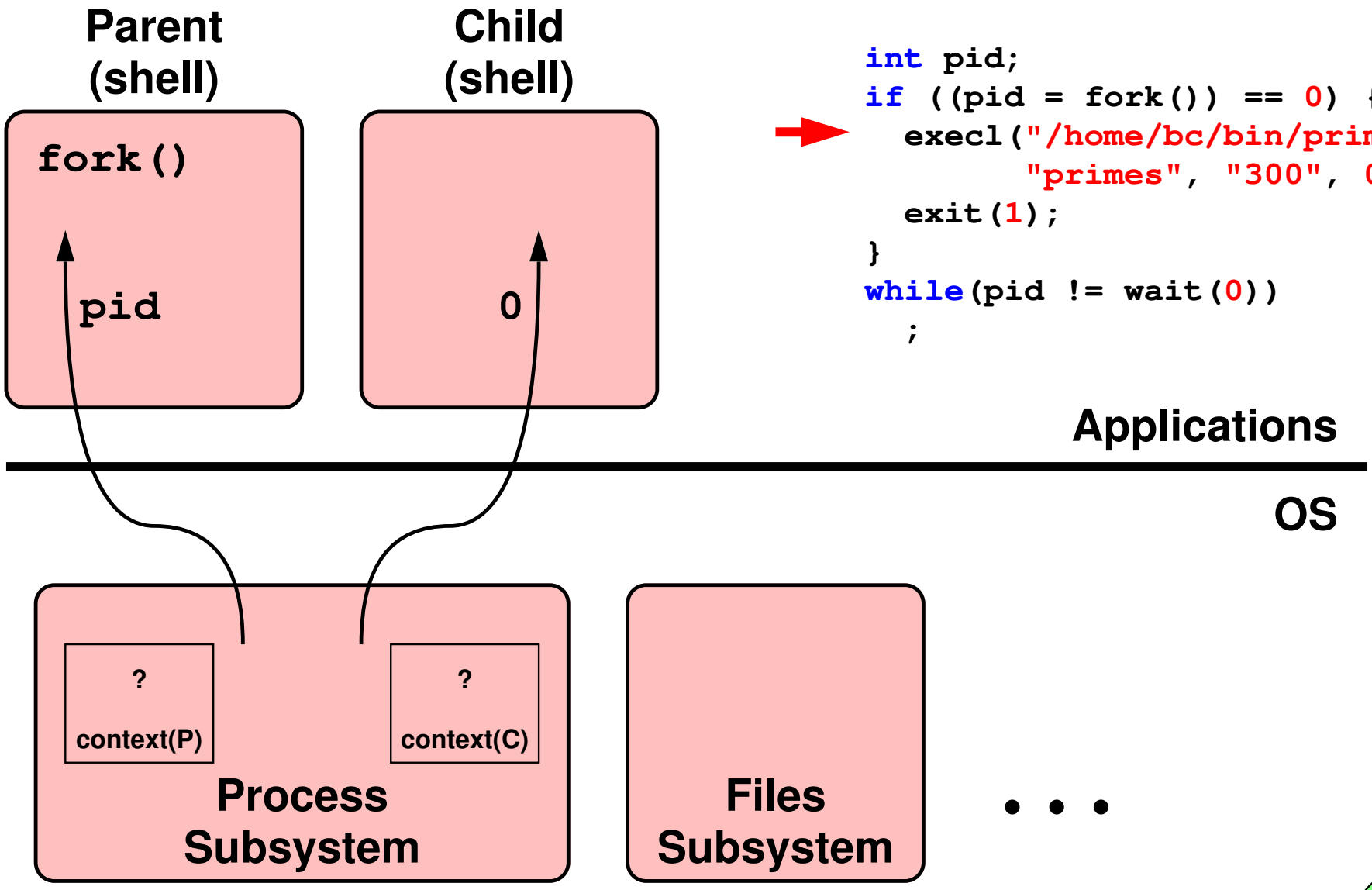




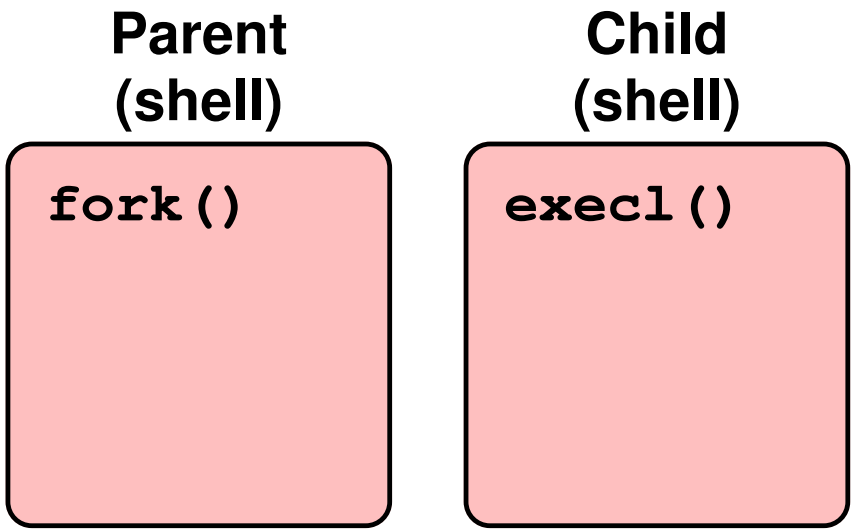
Put It All Together



Put It All Together



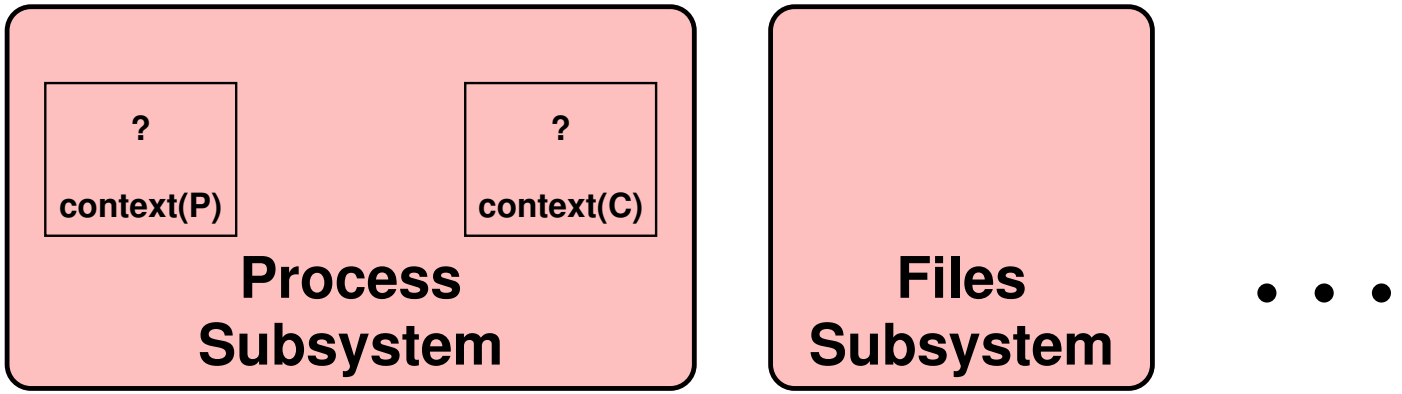
Put It All Together

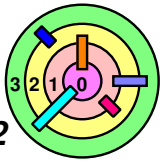



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

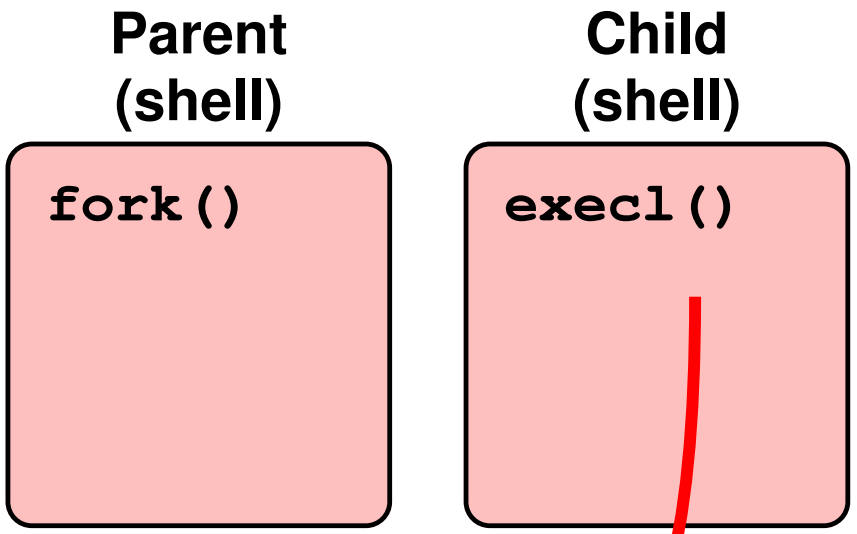
Applications

OS





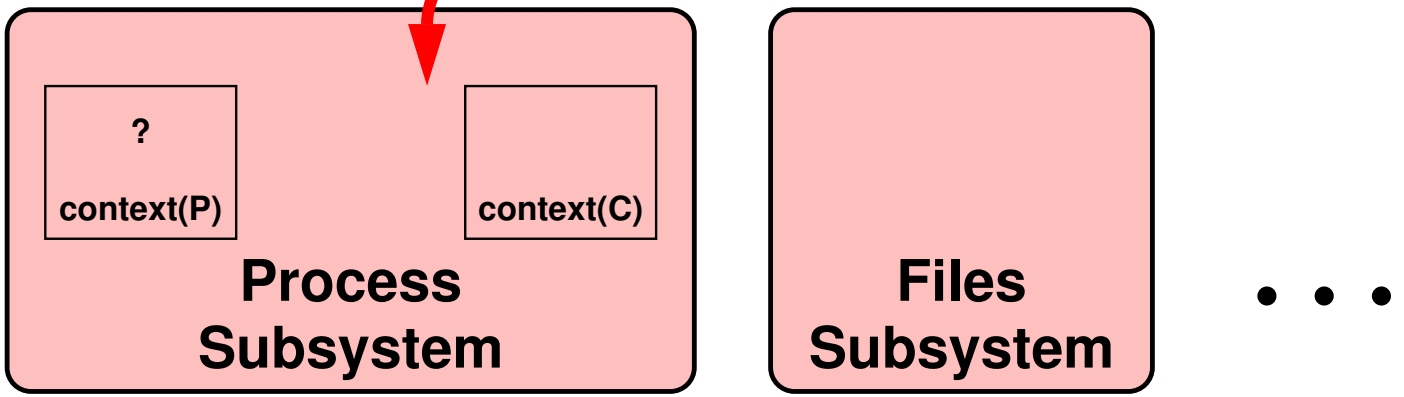
Put It All Together

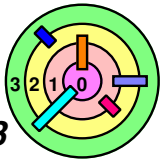



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

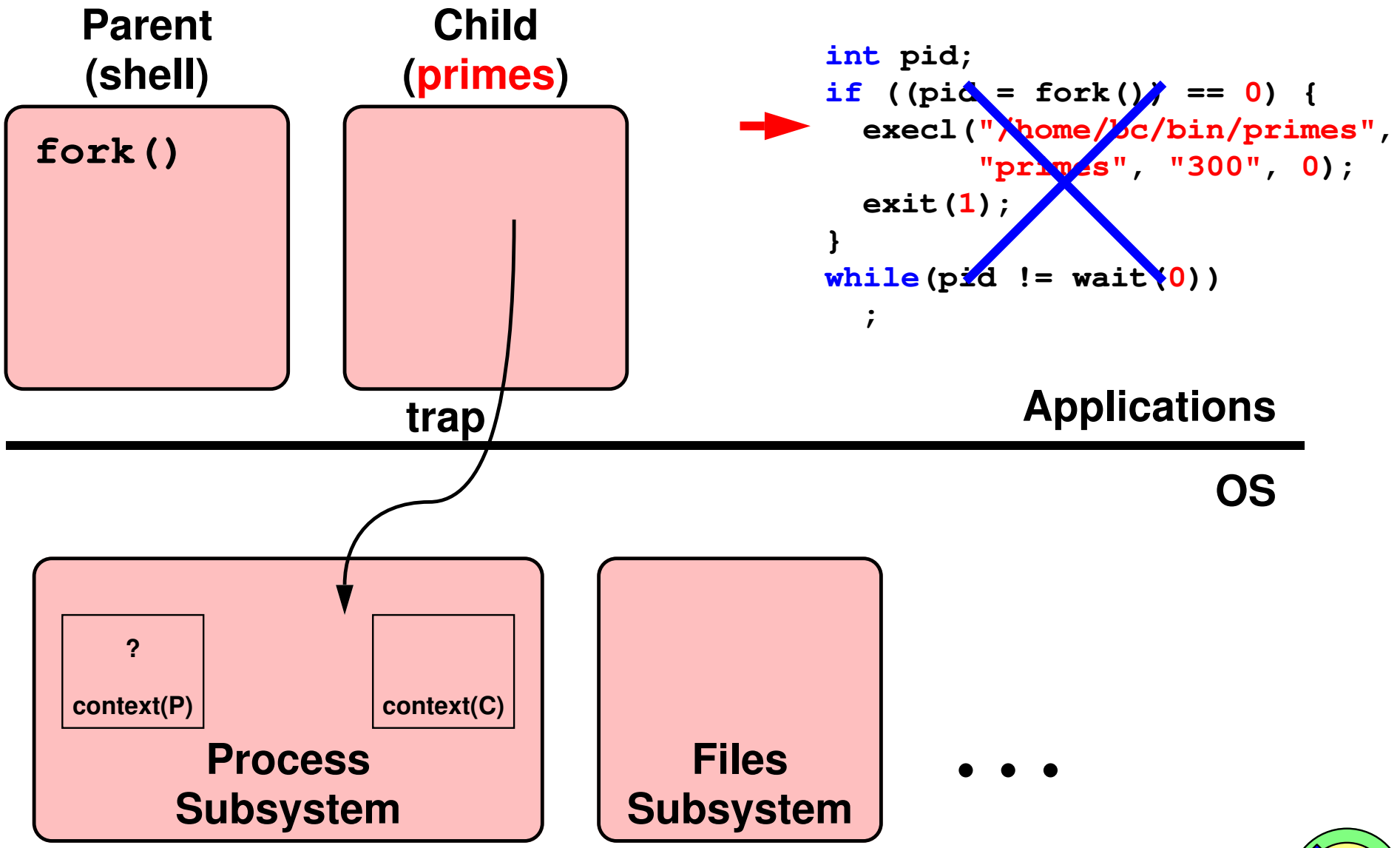
Applications

OS

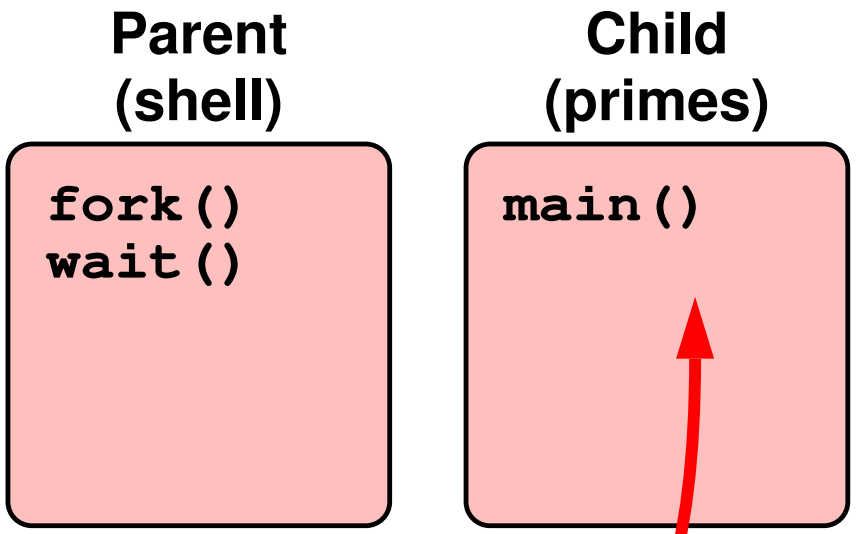




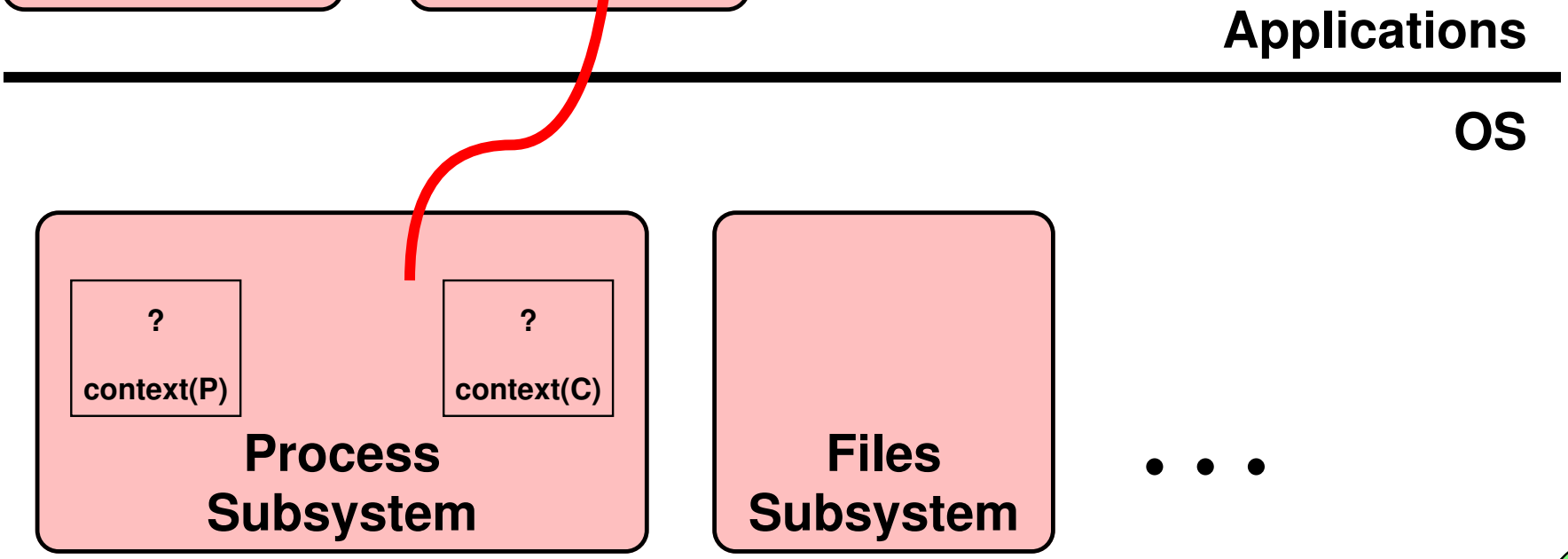
Put It All Together

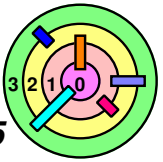


Put It All Together

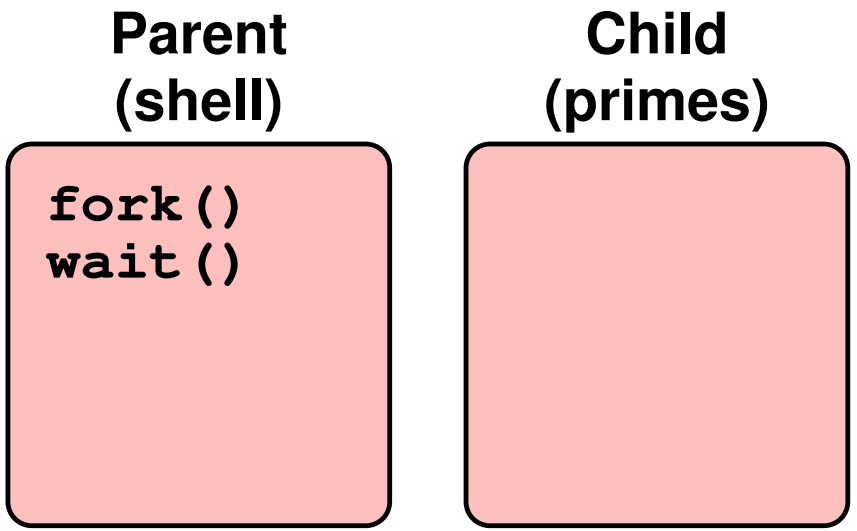



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```





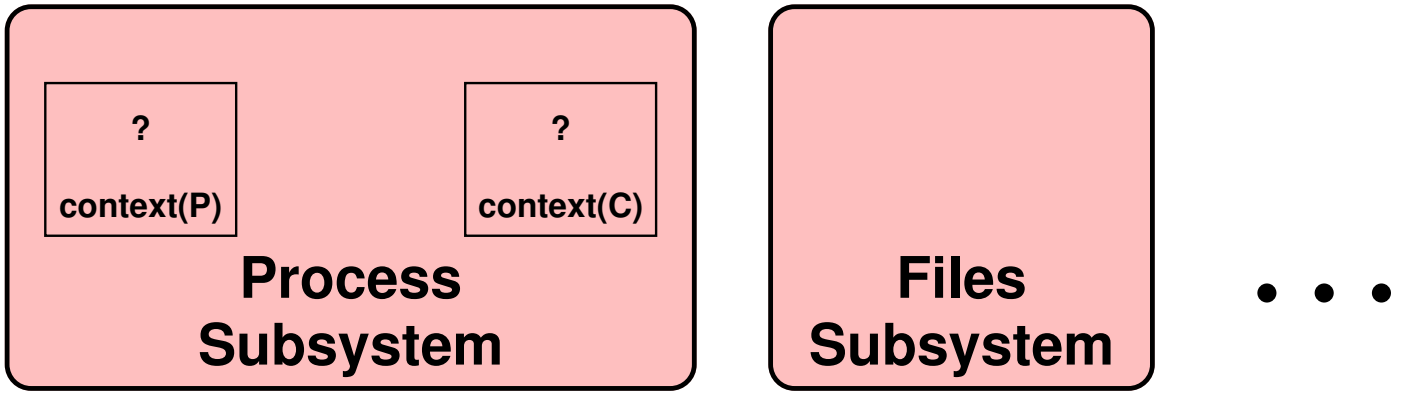
Put It All Together

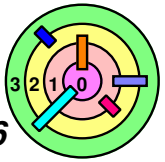



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

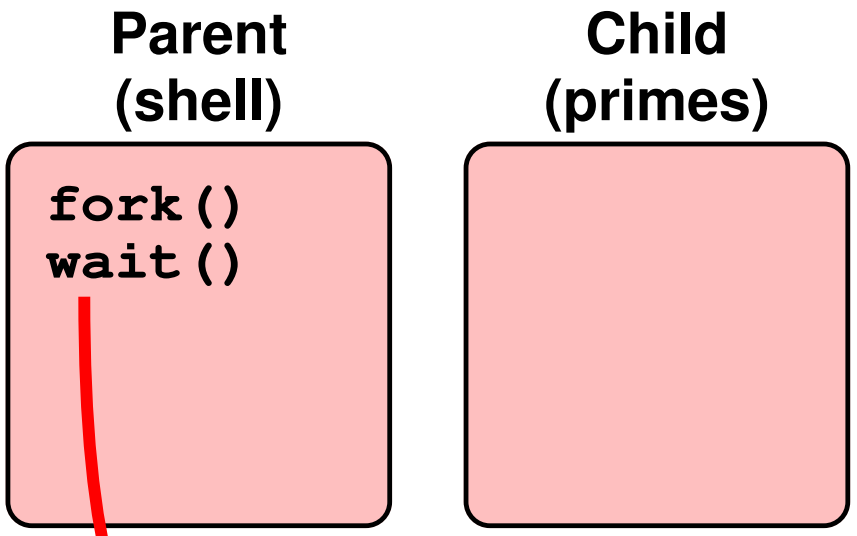
Applications

OS





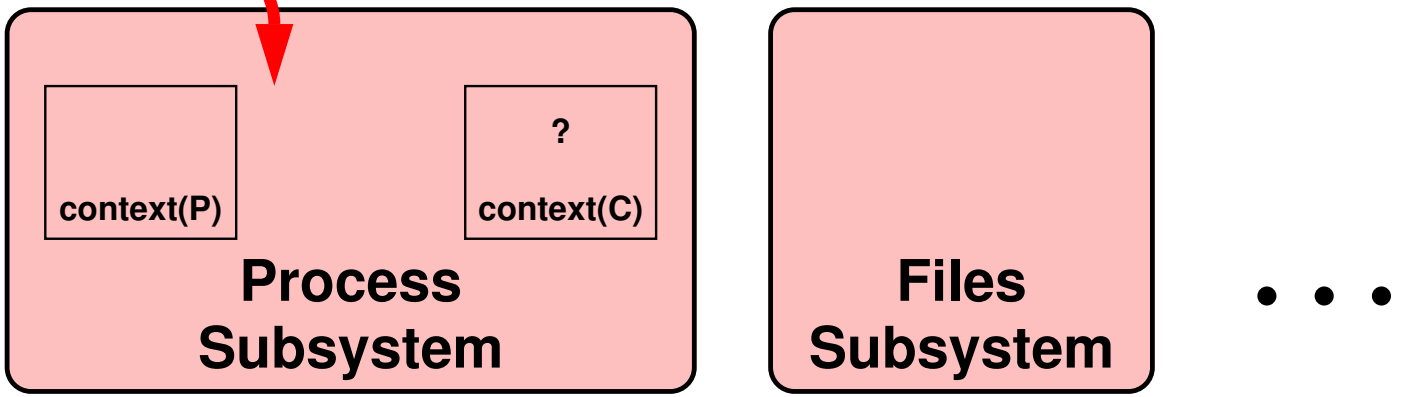
Put It All Together

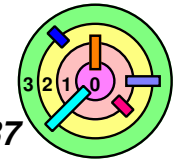



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

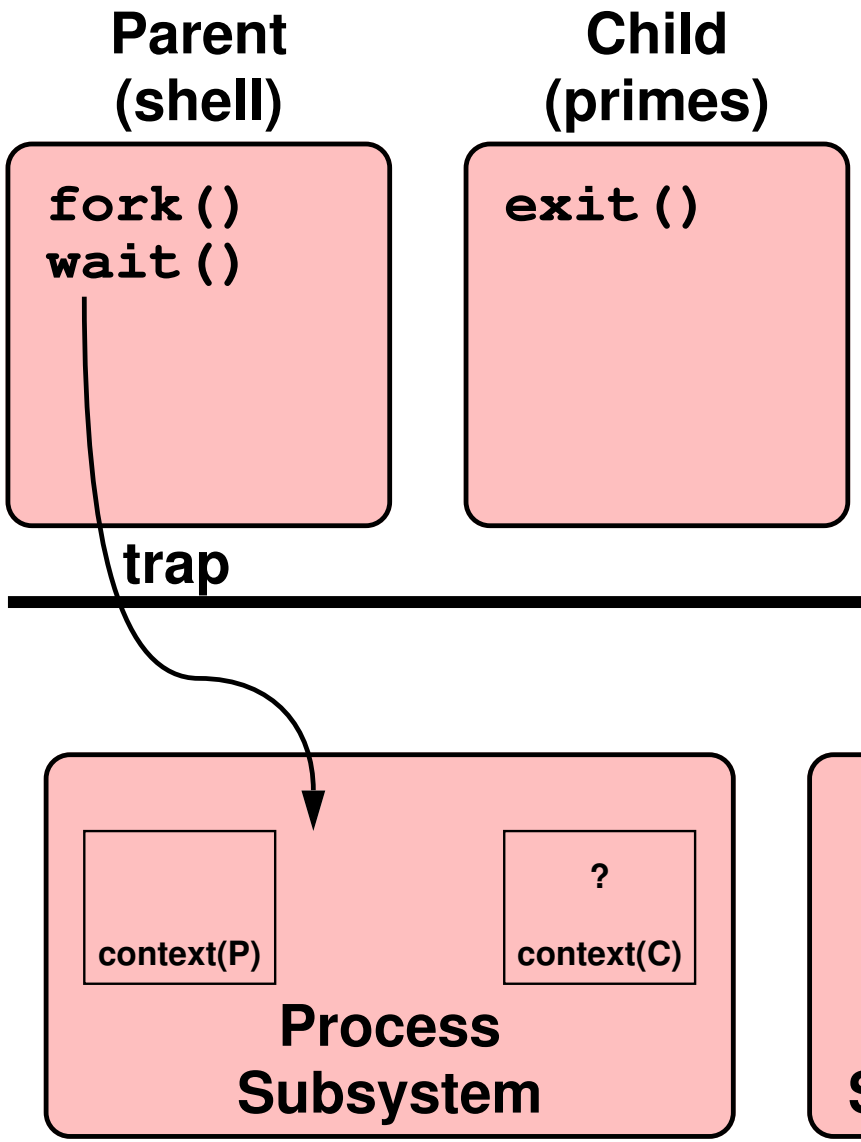
Applications

OS

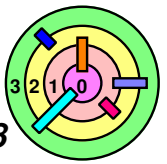




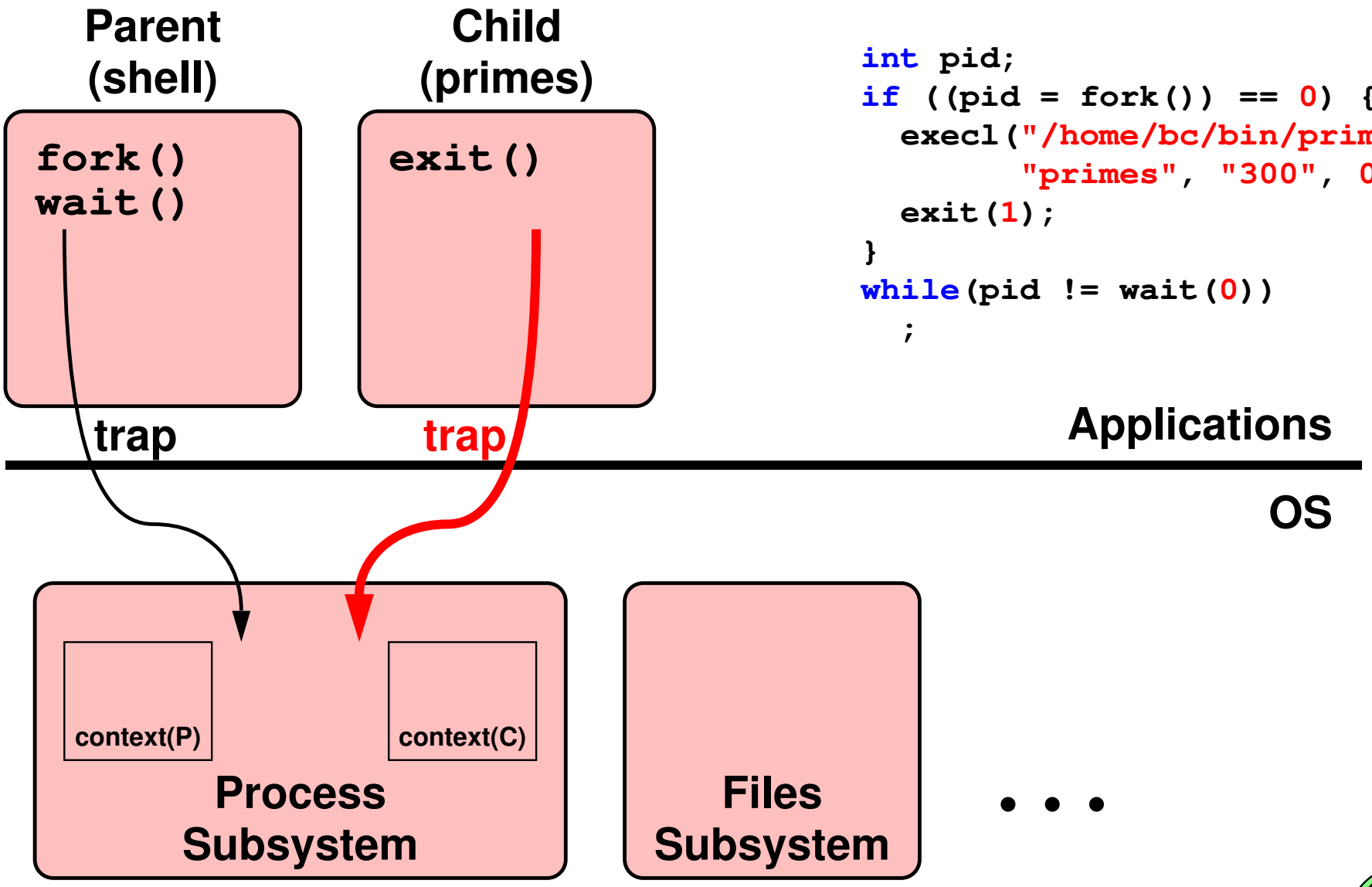
Put It All Together




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

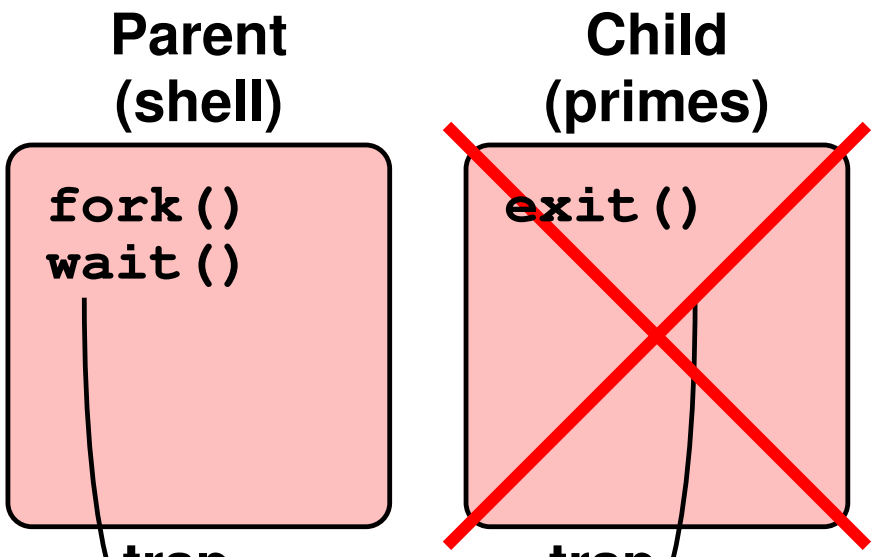


Put It All Together




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

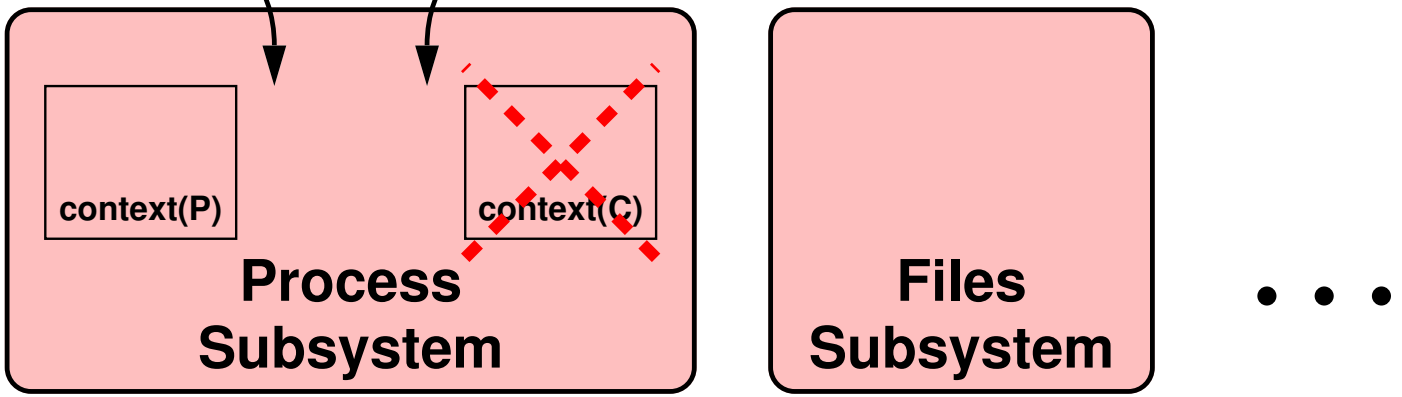
Put It All Together

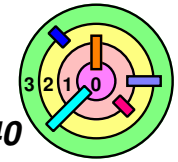



```
int pid;
if ((pid = fork()) == 0) {
  execl("/home/bc/bin/primes",
        "primes", "300", 0);
  exit(1);
}
while(pid != wait(0))
  ;
```

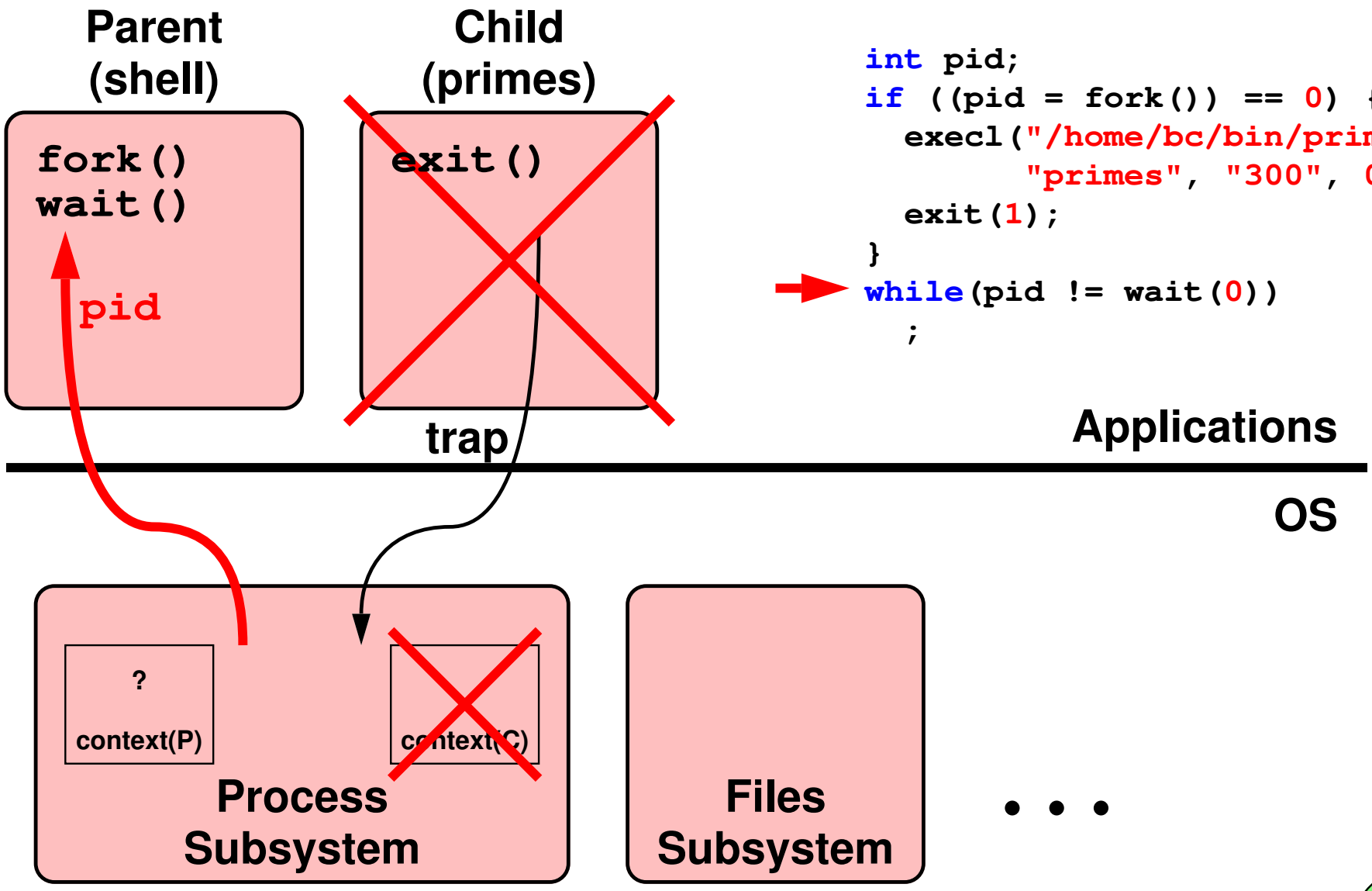
Applications

OS

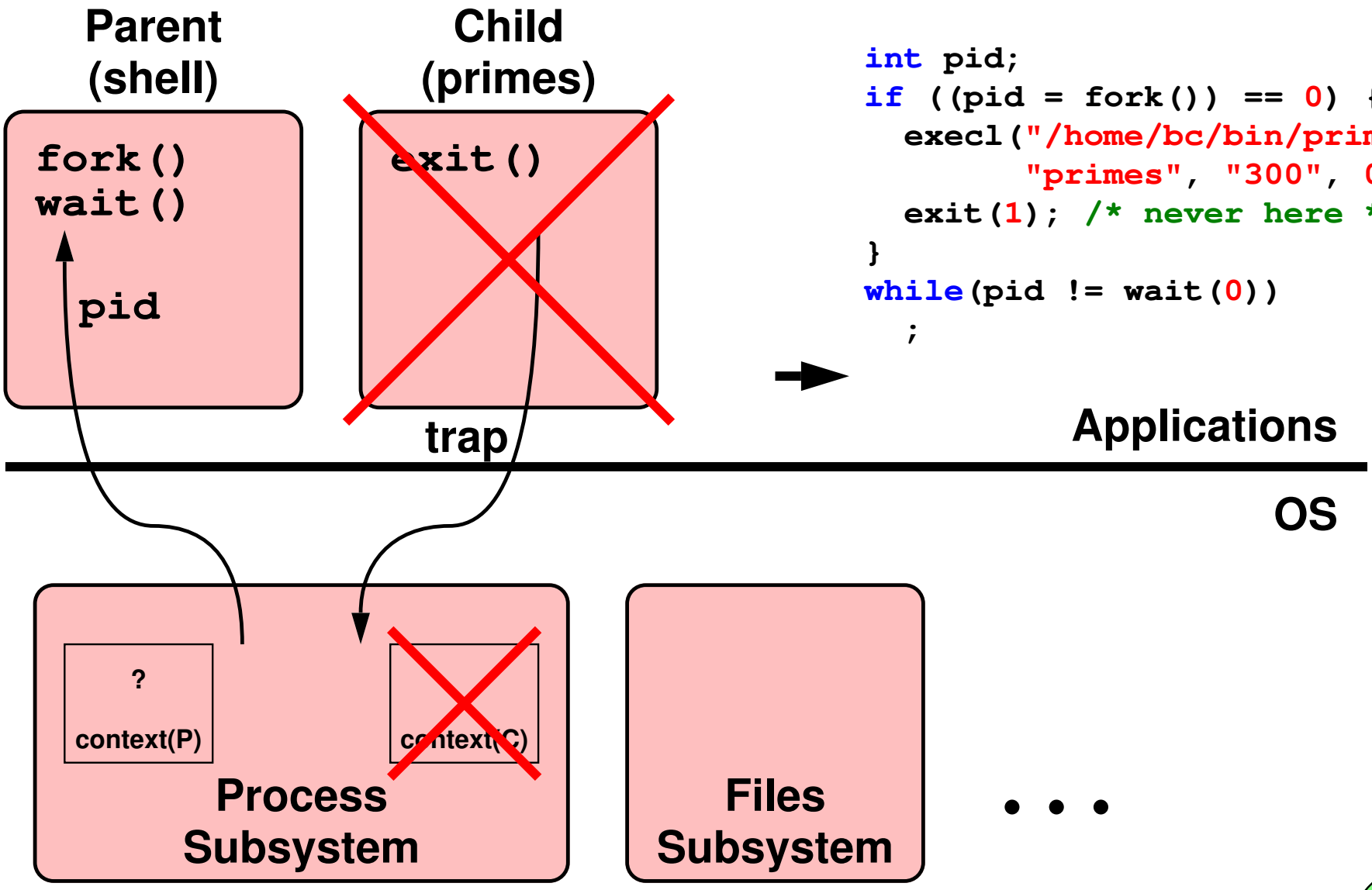




Put It All Together



Put It All Together

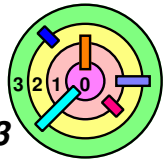


Unpredictable

Q: What does this code print?

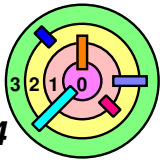
```
int main(int argc, char *argv[])
{
  int child_pid = fork();
  if (child_pid == 0) {
    printf("I am process #%d\n", getpid());
    exit(0);
  } else {
    printf("I am the parent of process #%d\n",
        child_pid);
  }
  return 0;
}
```

- unpredictable: parent could print first, or child could print first
  - on your machine, even if the order looks fixed after every run, there is no guarantee that the order would be the same the next time you run this program



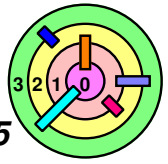
Questions?

- Can UNIX `fork()` return an error? Why?
- Can UNIX `exec()` return an error? Why?
- Can UNIX `wait()` ever return immediately? Why?



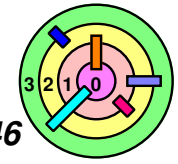
Implementing UNIX `fork()`

- Steps to implement UNIX `fork()`
  - create and initialize the process control block (PCB) in the kernel
  - create a new address space
  - initialize the address space with a copy of the entire contents of the address space of the parent
  - inherit the execution context of the parent (e.g., any open files)
  - inform the scheduler that the new process is ready to run



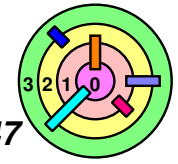
Implementing UNIX `execl()`

- Steps to implement UNIX `execl()`
  - load the program into the current address space
  - copy arguments into memory in the address space
  - initialize the hardware context to start execution at `start()`
- An exec system call does not create a new process



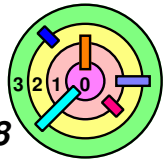
exec and `wait()`

- Call to `wait()` is optional
  - command shell can run a program/job *in the background* (signified by appending "`&`" to the command line)
    - *in the background* means not waiting for it to die
    - use the "`jobs`" shell command to list all jobs running in the background for your command shell
    - use the "`fg`" shell command to bring a background process to the foreground (and wait for it to die)
    - use the "`%2`" shell command to bring background job `[2]` to the foreground (and wait for it to die)
    - press <Ctrl+Z> to suspend the foreground job
      - use the "`bg`" shell command to run the suspended job in the background
- In Windows, you can call `WaitForSingleObject()` to wait for a process to die or a thread to die



UNIX Notifications

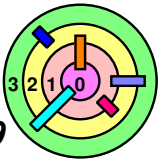
- In UNIX, a notification is sent by sending a ***signal***
  - signals are used for a variety of purposes
    - terminate an application
    - suspending an application temporarily for debugging
    - resuming after a suspension
    - timer expiration
    - keyboard interrupts
      - <Ctrl+C>
      - <Ctrl+Z>
    - etc.
  - if an application does not specify a signal handler, the kernel implements a standard one on its behalf



CS 350
PA2: Kernel Level Threads

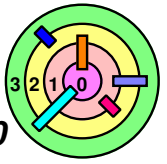
Bill Cheng

http://merlot.usc.edu/william/usc/



PA2

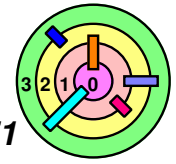
- Set up a standard 32-bit Ubuntu 16.04 system
  - download xv6 for PA2
- Part 1 - add threads to the kernel
  - `fork()`, `exec()`, `exit()`
- Part 2 - thread system calls
  - `kthread_create()`, `kthread_id()`, `kthread_exit()`, `kthread_join()`



Download XV6 For PA2

- Follow the instructions on the PA2 spec
- Open a *terminal* and type the following

```
cd
cd cs350
mkdir pa2
cd pa2
wget --user=USERNAME --password=PASSWORD \
  http://merlot.usc.edu/cs350-m25/programming/pa2/xv6-pa2-src.tar.gz
tar xvf xv6-pa2-src.tar.gz
cd xv6-pa2-src
```



Submission

- Which files do you need to modify?
  - open a terminal and type the following:

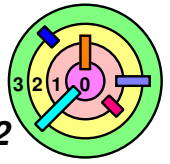
```
pwd
cd cs350/pa1/xv6-pa2-src
make -n pa2-submit
```

    - you should see:

```
tar cvzf pa2-submit.tar.gz \
        Makefile \
        pa2-README.txt \
        proc.c \
        proc.h \
        syscall.c \
        sysproc.c \
        kthread.h \
        exec.c
```

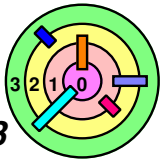
      - I don't think you need to modify Makefile at all

  - these are the only files are are supposed to submit
    - if you submit additional files, the grader will have to delete them before grading
    - if you submit binary files, points will be deducted



Part 1: Add Threads To The Kernel

- Change the implementation of some existing system calls
  - `fork()`, `exec()`, `exit()`
    - `fork()`, and `exit()` are in "`proc.c`"
    - `exec()` is in "`exec.c`"
  - note: some functions might not need changes (you need to pick which ones to change)

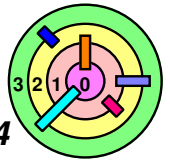


Background: `growproc()`

`growproc()` is responsible for retrieving more memory when the process asks for it

```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

- to ***access any PCB***, must acquire the `ptable.lock` spinlock
- need to synchronize accesses to `proc->sz`

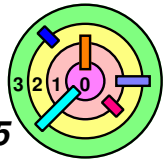


Background: `growproc()`

- `growproc()` is responsible for retrieving more memory when the process asks for it

```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

  - always release locks before return statement if it is not released previously

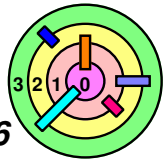


Background: `growproc()`

- `growproc()` is responsible for retrieving more memory when the process asks for it

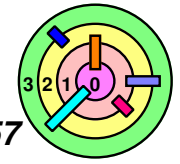
```
// Grow current process's memory
//     by n bytes.
// Return 0 on success,
//     -1 on failure.
int
growproc(int n)
{
  uint sz;
  acquire(&ptable.lock);
  sz = proc->sz;
  if (n > 0){
    if ((sz = allocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  } else if (n < 0){
    if ((sz = deallocuvm(
        proc->pgdir, sz,
        sz + n)) == 0){
      release(&ptable.lock);
      return -1;
    }
  }
  proc->sz = sz;
  switchuvm(proc);
  release(&ptable.lock);
  return 0;
}
```

- you might need to think more about synchronization and find where to put functions/methods, locks, etc.



`fork()`

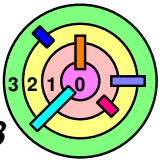
- `fork()` should duplicate only the calling thread, if other threads exist in the process they will notexist in the new process
- Questions to ask:
  - are there any conflicts between shared variables?
  - do we need to kill any threads after calling fork?
  - is the acquired the lock enough for synchronization or should we put more locks?



exit()

- `exit()` should kill the process and all of its threads, remember while a single threads executing `exit()`, others threads of the same process might still be running

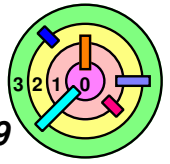
```
kill_all();

// jump into the scheduler, never to return
thread->state = TINVALID;
proc->state = ZOMBIE;
sched();
panic("zombie exit");
```



kill_all()

We have to create a `kill_all()` function to kill all the alive threads:

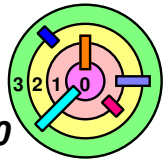
```
kill_all():
  create thread pointer *t
  for each thread t:
    if (thread t is not the current thread and
        not running and not unused) then
      make t a zombie
    end-if
  end-for
  make current thread zombie
  kill process
```



exec()

- The thread performing exec should "tell" other threads of the same process to destroy themselves and only then complete the `exec()` task
- modify `kill_all()` method and create `kill_others()`
  - `kill_others()` kills all alive threads but itself

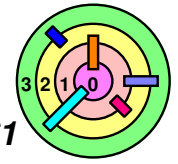
```
kill_others():
  create thread pointer *t
  for each thread t:
    if (thread t is not the current thread and
        not running and not unused) then
      make t a zombie
    end-if
  end-for
```



Part 2: Thread System Calls

- Implement thread API for kernel
  - `kthread_create()`, `kthread_id()`, `kthread_exit()`, `kthread_join()`
  - you will implement these functions in "`proc.c`" and add the following to "`kthread.h`"

```
int kthread_create(void*(*start_func)(),
                   void* stack, int stack_size);
int kthread_id();
void kthread_exit(void);
int kthread_join(int thread_id);
```



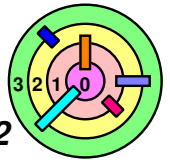
Changing Thread States

```
t->state = TZOMBIE;
```

- in "proc.h"

```
enum threadstate {
  TUNUSED,
  TEMBRYO,
  TSLEEPING,
  TRUNNABLE,
  TRUNNING,
  TZOMBIE,
  TINVALID
};
```

```
enum procstate {
  UNUSED,
  USED,
  ZOMBIE
};
```



Changing Thread States

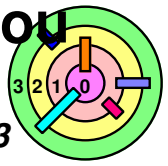
`t->tid != thread->tid;`

- `thread`, `proc`, and `cpu` are global variables that point to the current thread, the current process, and the current CPU
- in "`proc.h`"

```
struct thread {
  int tid;                          // Thread ID
  enum threadstate state;           // thread state
  char *kstack;                     // Bottom of kernel stack for this thread
  struct proc *parent;              // Parent process
  struct trapframe *tf;             // Trap frame for current syscall
  struct context *context;          // swtch() here to run process
  void *chan;                       // If non-zero, sleeping on chan
  int killed;                       // If non-zero, have been killed
};
```

⇨ Read the code in `allocthread()` to see how every field is initialized

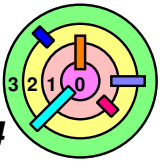
- e.g., since the trap frame is the bottom of the kernel stack, you don't need to allocate memory for the trap frame



How To Loop Through Threads?

Look at `allocthread()` in "`proc.c`"

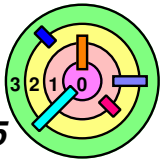
```
struct thread*
allocthread(struct proc * p)
{
  struct thread *t;
  for(t = p->threads; found != 1 && t < &p->threads[NTHREAD]; t++) {
    ...
  }
  ...
}
```



How To Loop Through Processes?

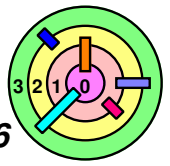
Look at `allocproc()` in "`proc.c`"

```
struct thread*
allocproc()
{
  struct proc *p;
  struct thread *t;

  for(p = ptable.proc; && p < &ptable.proc[NPROC]; p++) {
    ...
  }
  ...
}
```



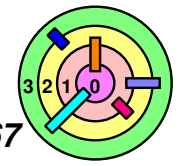
kill_all()

```
kill_all():
  create thread pointer *t
  for each thread t: // loop through threads
    if (t is not current thread and not running
        and not unused) then // check its state
      make t a zombie // change its state
    end-if
  end-for
  make current thread zombie // find current thread
                             // and change its state
  kill process // proc->killed = 1
```



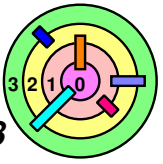
kthread_create()

- Calling `kthread_create()` will create a new thread within the context of the calling process
  - the newly created thread state will be TRUNNABLE
  - the caller of `kthread_create()` must allocate a user stack for the new thread to use (it should be enough to allocate a single page i.e., 4K for the thread stack)
  - this does not replace the kernel stack for the thread
- `start_func` is a pointer to the entry function, which the thread will start executing
  - upon success, the identifier of the newly created thread is returned
  - in case of an error, a non-positive value is returned



kthread_create()

- The kernel thread creation system call on real Linux does not receive a user stack pointer
  - in Linux the kernel allocates the memory for the new thread stack
  - you will need to create the stack in user mode and send its pointer to the system call in order to be consistent with current memory allocator of xv6

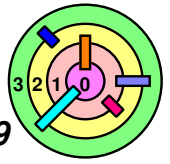


kthread_create()

```
kthread_create(void* (*start_func)(), void* stack, int stack_size):

  create a thread pointer
  allocate the thread using allocthread() function
  check if t is 0 // allocated correctly?
    if not, return -1
  else
    copy current thread's trap frame
    find stack address of the thread using stack pointer given parameter
    make stack pointer inside trap frame stack address + stack size
    update base pointer inside trap frame as stack pointer
    find address of the start function which is given in parameter
    make instruction pointer inside trap frame start address
    return tid
```

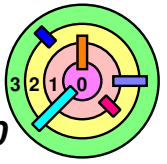
- stack pointer: `t->tf->esp`
- base pointer: `t->tf->ebp`
- instruction pointer: `t->tf->eip`

⇨ **Note: the above is not the only way to create a thread**



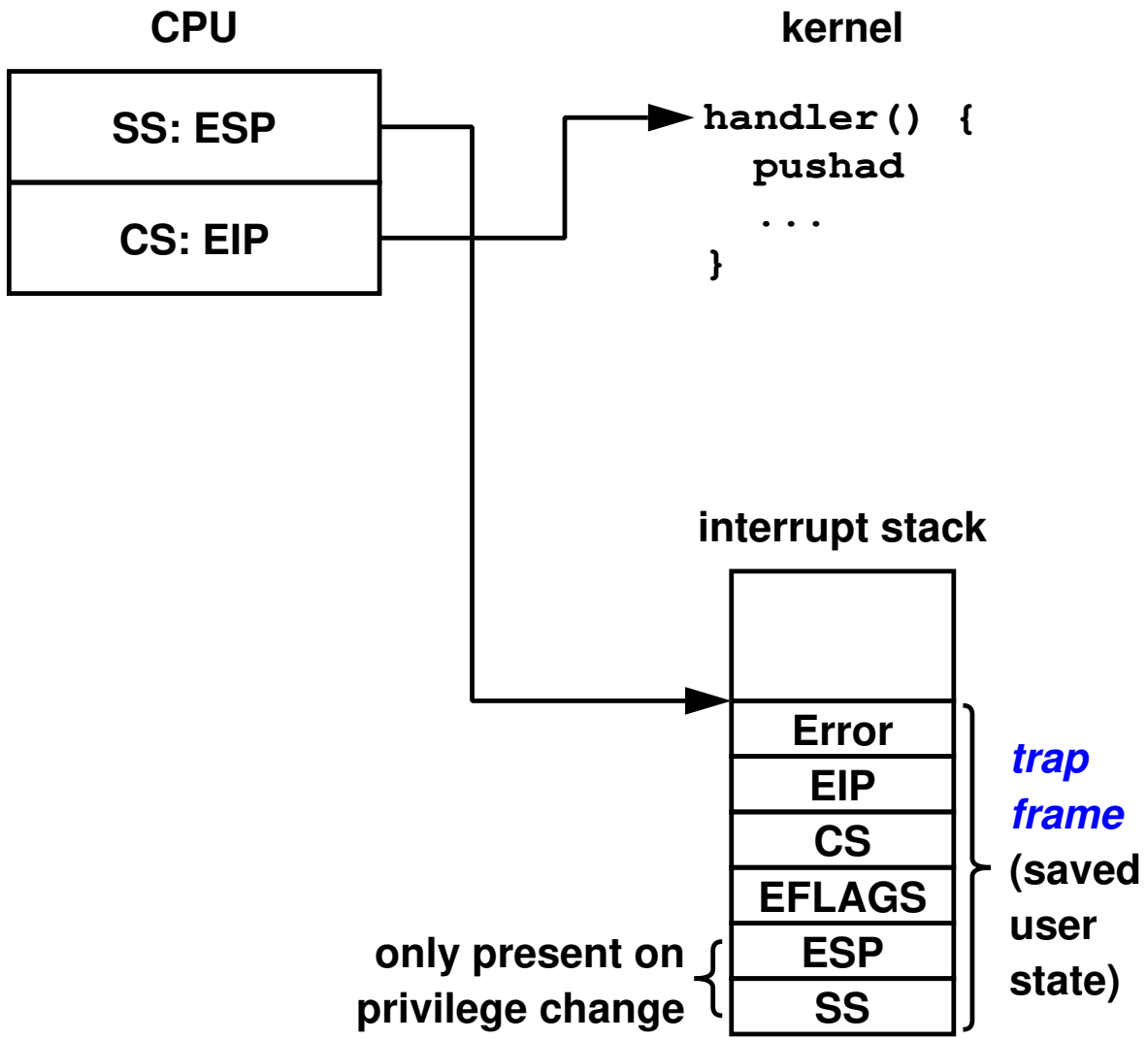
esp, eip, ebp

- `t->tf->esp`
  - we have to change the stack address of new thread
  - so we use given parameter to make stack address different than current thread
  - add given stack's address with stack size to find where to put stack pointer
- `t->tf->ebp`
  - initially base pointer and stack pointer point the same place
- `t->tf->eip`
  - instruction pointer points the instructions which will be implemented by thread
  - this pointer has to point stack function at the beginning



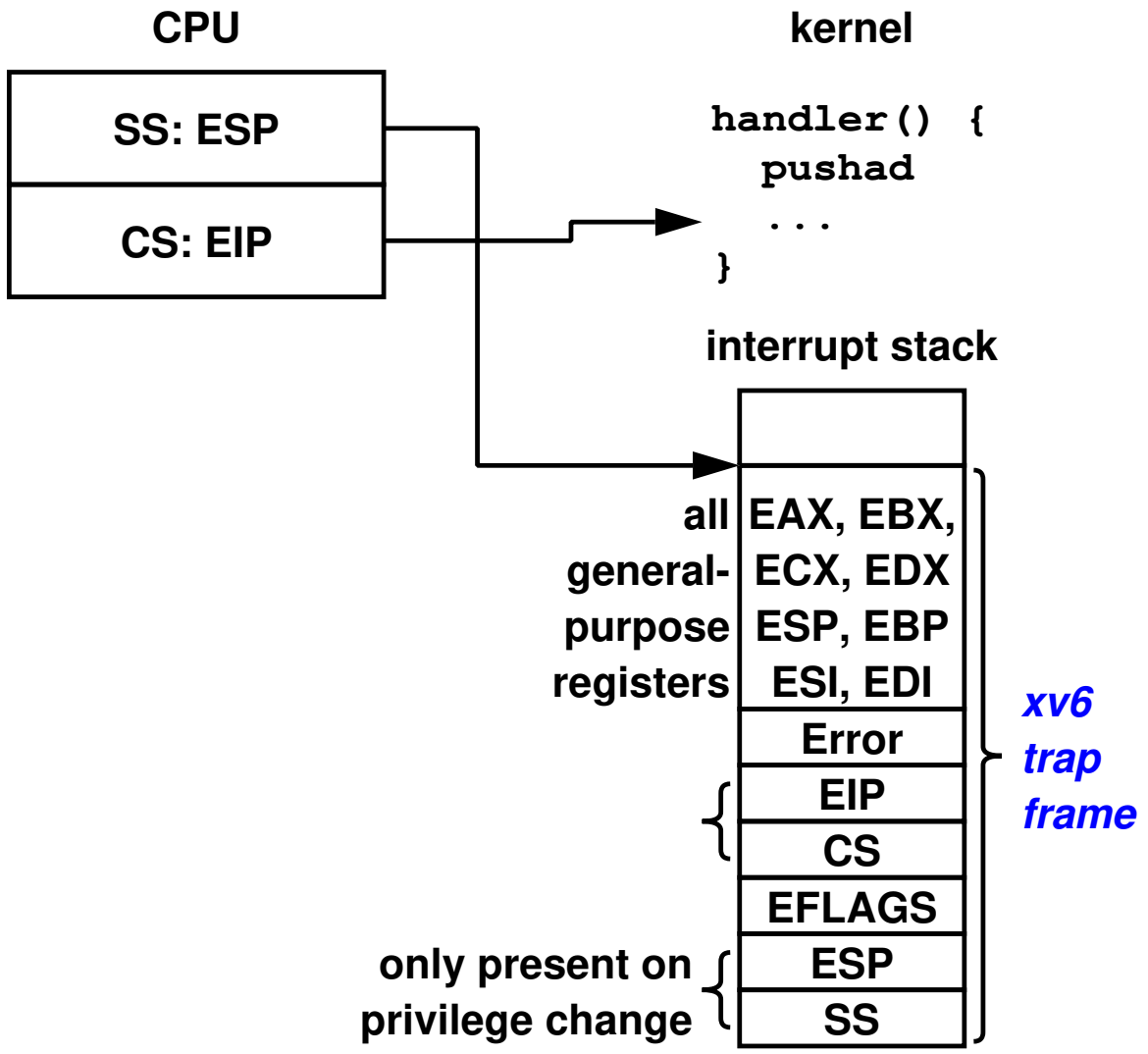
More Information In Ch 3 Of XV6 Book

- Figure 3.1 of xv6 book (with low address on top and high address on the bottom) shows the kernel stack after an `INT` instruction
  - also in Ch 2 slides for the x86 CPU



More Information In Ch 3 Of XV6 Book

- Figure 3.1 of xv6 book (with low address on top and high address on the bottom) shows the kernel stack after an `INT` instruction
  - also in Ch 2 slides for the x86 CPU

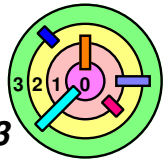


kthread_id()

- Easiest function to implement in PA2
  - upon success, this function returns the caller thread's id
  - in case of error, a non-positive error identifier is returned
  - remember, thread id and process id are not the same thing

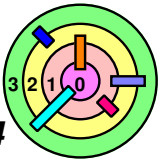
```
kthread_id():

  if process and thread exists
    return t->tid
  else
    return -1
```

- Note: this is not the only way to return a thread id



`kthread_exit()`

- This function terminates the execution of the calling thread
  - if called by a thread (even the main thread) while other threads exist within the same process, it shouldn't terminate the whole process
  - if it is the last running thread, process should terminate
  - each thread must explicitly call `kthread_exit()` in order to terminate normally



kthread_exit()

```
kthread_exit():

  create a thread pointer
  create a found flag
  loop through all threads to find another thread running
    if t is not current thread // because calling thread is current
      if t is not unused, not a zombie, and not invalid
        make found flag true
        break // only one running is enough

  if found // I am not the last thread in my process
    wakeup all waiting using wakeup1() // read wakeup1() code
  else // found flag is false, therefore, I'm the last thread
    exit()
  make this thread zombie
  call sched() to schedule another thread
```

- Note: the above is not the only way to exit a thread
- also, this is not a complete pseudocode
  - you have to add locks if necessary