COS 217: Introduction to Programming Systems

Processes

PRINCETON UNIVERSITY

Agenda

Processes

- Illusion: Private address space
- Illusion: Private control flow

Process management in C

- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control

- Signals
- Alarms

Processes

Program

- Executable code
- A static entity

Process

- An instance of a program in execution
- A dynamic entity: has a time dimension
- Each process runs one program
    - E.g. the process with Process ID 12345 might be running emacs
- One program can run in multiple processes
    - E.g. PID 12345 might be running emacs, and PID 23456 might also be running emacs – for the same user or for a different user

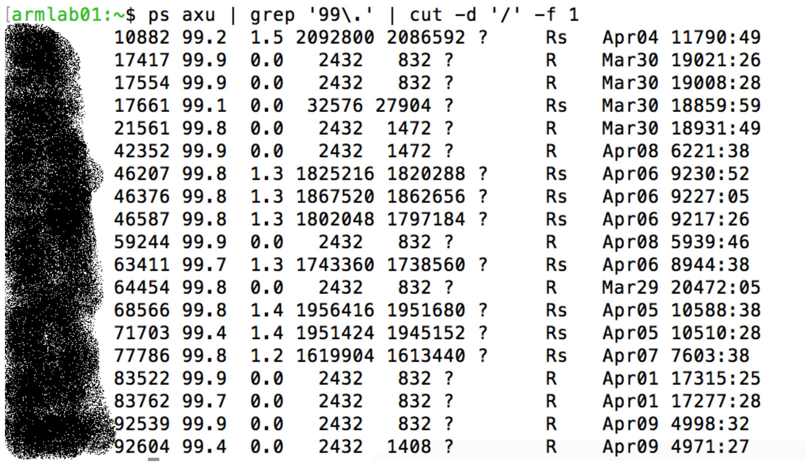
Processes

Program

- Executable code
- A static entity

Process

- An instance of a program in execution
- A dynamic entity: has a time dimension
- Each process runs one program
  - E.g. the process with Process ID 12345 might be running emacs
- One program can run in multiple processes
  - E.g. PID 12345 might be running emacs, and PID 23456 might also be running emacs – for the same user or for different users



```
armlab01:~$ ps axu | grep '99\.' | cut -d '/' -f 1
[redacted] 10882 99.2  1.5 2092800 2086592 ?     Rs   Apr04 11790:49
           17417 99.9  0.0   2432    832 ?       R    Mar30 19021:26
           17554 99.9  0.0   2432    832 ?       R    Mar30 19008:28
           17661 99.1  0.0  32576  27904 ?       Rs   Mar30 18859:59
           21561 99.8  0.0   2432   1472 ?       R    Mar30 18931:49
           42352 99.9  0.0   2432   1472 ?       R    Apr08 6221:38
           46207 99.8  1.3 1825216 1820288 ?     Rs   Apr06 9230:52
           46376 99.8  1.3 1867520 1862656 ?     Rs   Apr06 9227:05
           46587 99.8  1.3 1802048 1797184 ?     Rs   Apr06 9217:26
           59244 99.9  0.0   2432    832 ?       R    Apr08 5939:46
           63411 99.7  1.3 1743360 1738560 ?     Rs   Apr06 8944:38
           64454 99.8  0.0   2432    832 ?       R    Mar29 20472:05
           68566 99.8  1.4 1956416 1951680 ?     Rs   Apr05 10588:38
           71703 99.4  1.4 1951424 1945152 ?     Rs   Apr05 10510:28
           77786 99.8  1.2 1619904 1613440 ?     Rs   Apr07 7603:38
           83522 99.9  0.0   2432    832 ?       R    Apr01 17315:25
           83762 99.7  0.0   2432    832 ?       R    Apr01 17277:28
           92539 99.9  0.0   2432    832 ?       R    Apr09 4998:32
           92604 99.4  0.0   2432   1408 ?       R    Apr09 4971:27
```

```
armlab02:~$ cat /proc/sys/kernel/pid_max
98304
```

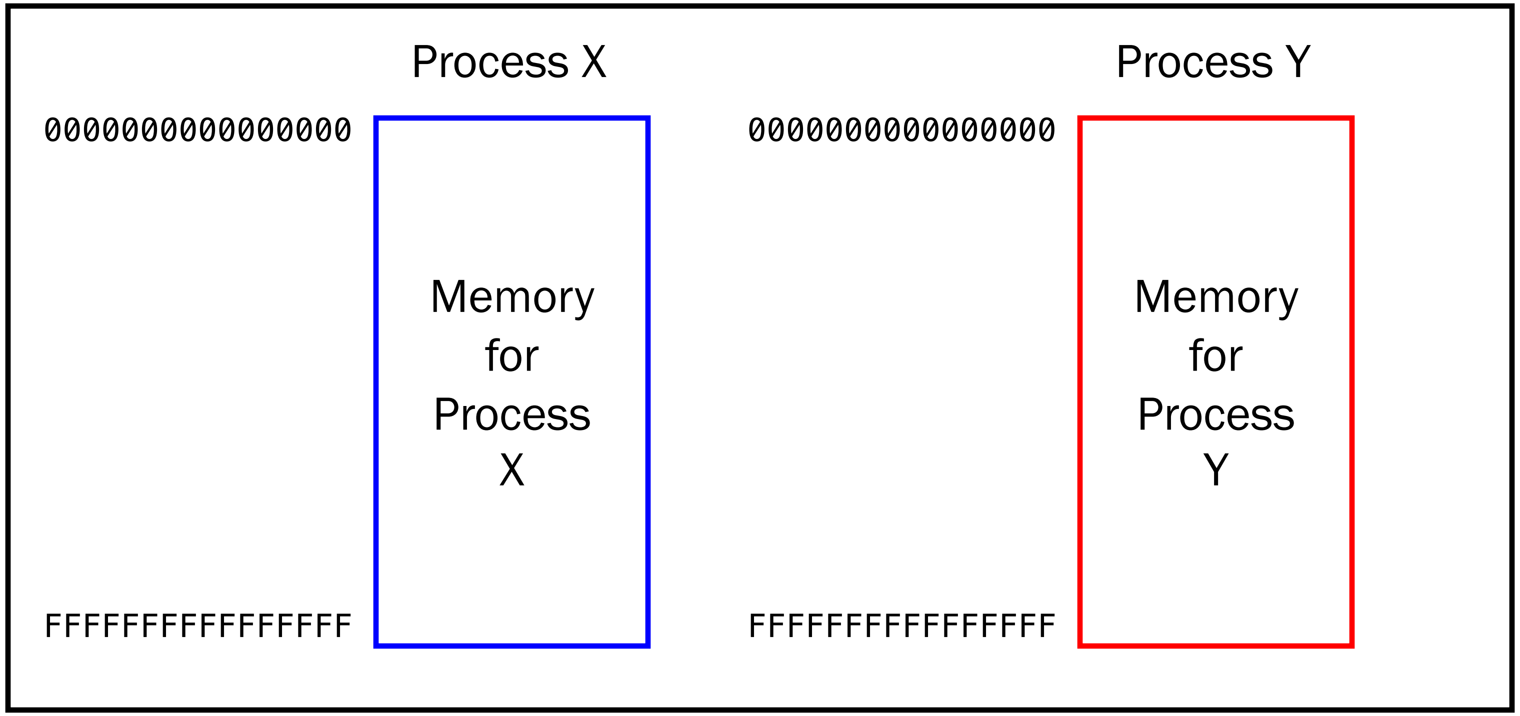
Processes Significance

Process abstraction provides two key illusions:

- Processes believe they have a *private address space*
- Processes believe they have *private control flow*

Process is a profound abstraction in computer science

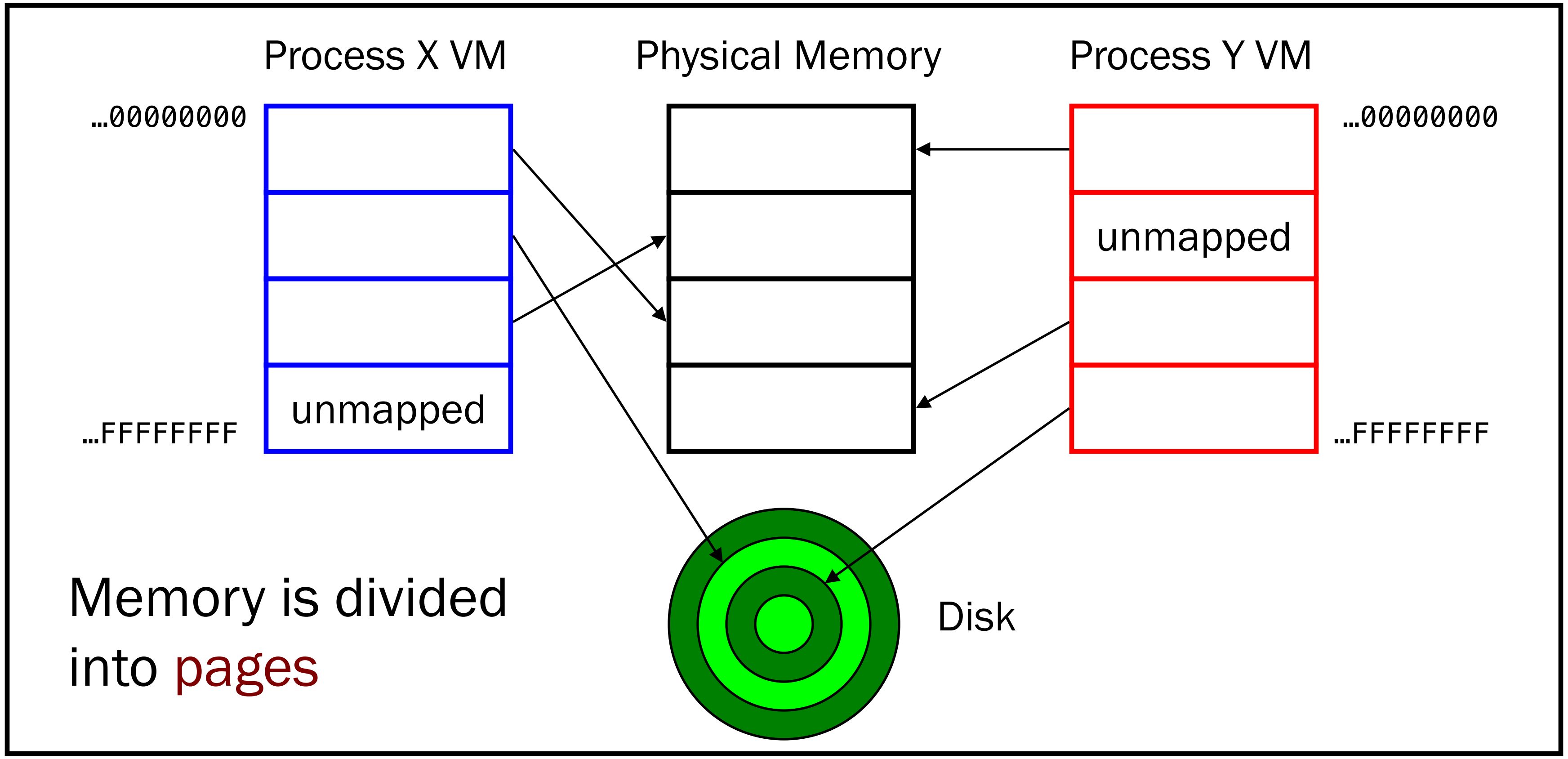
Private Address Space: Illusion



Hardware and OS give each application process
the illusion that it is the only process using memory

- Enables multiple simultaneous instances of one program!

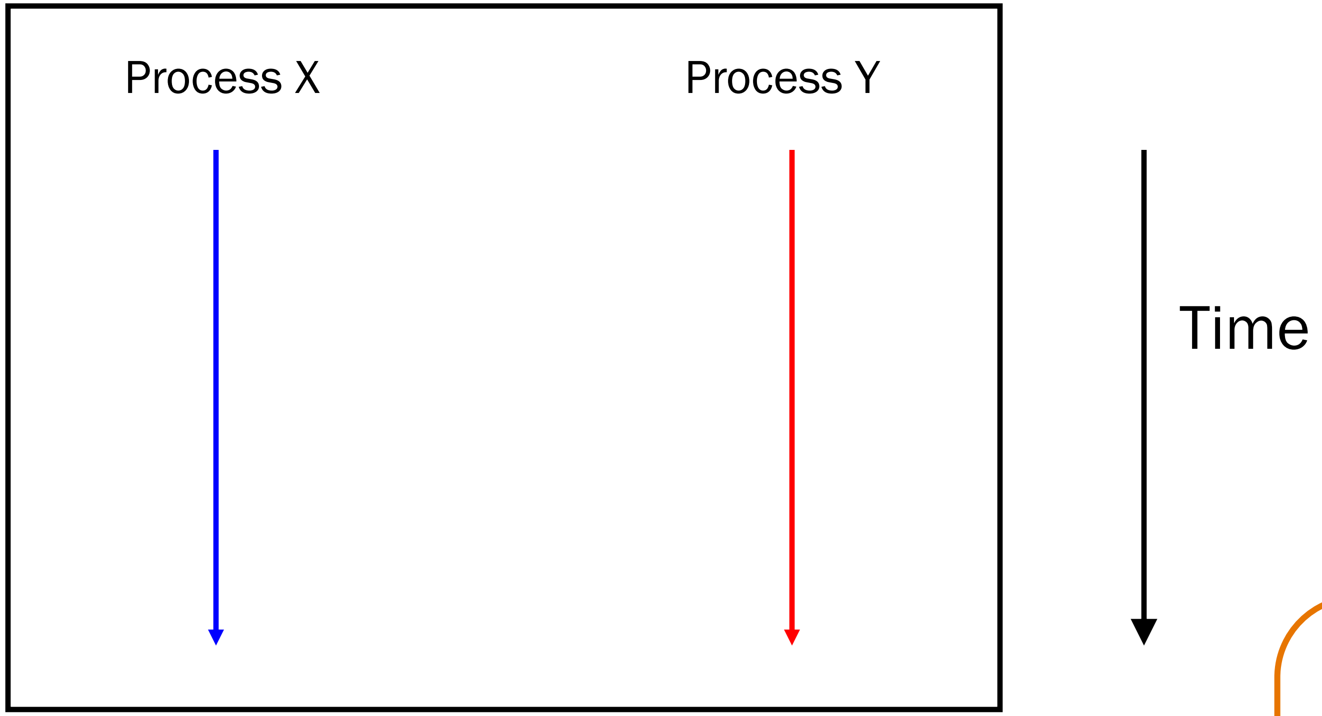
Private Address Space: Reality



Memory is divided into pages

All processes use the same physical memory.
Hardware and OS provide programs with
a virtual view of memory, i.e. virtual memory (VM)

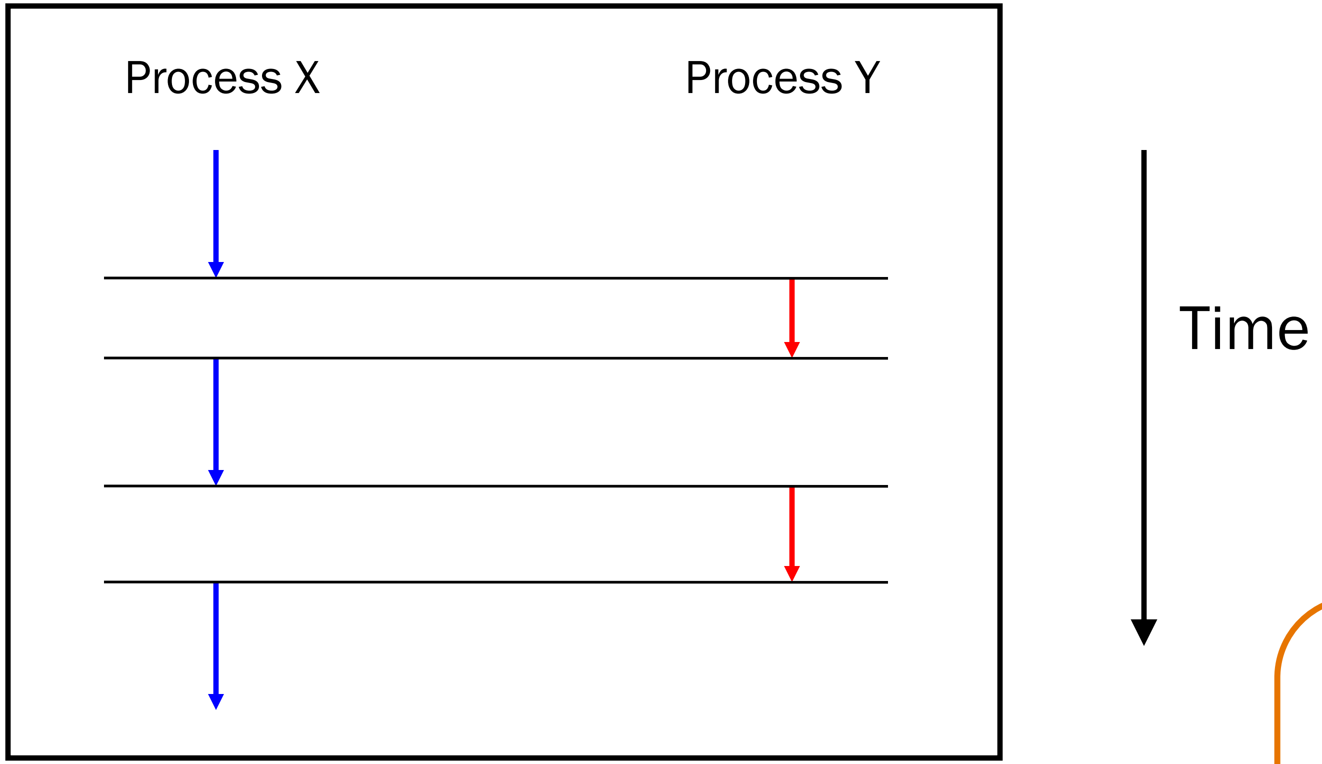
Private Control Flow: Illusion



Simplifying assumption: only one CPU / core

Hardware and OS give each application process the illusion that it is the only process running on the CPU

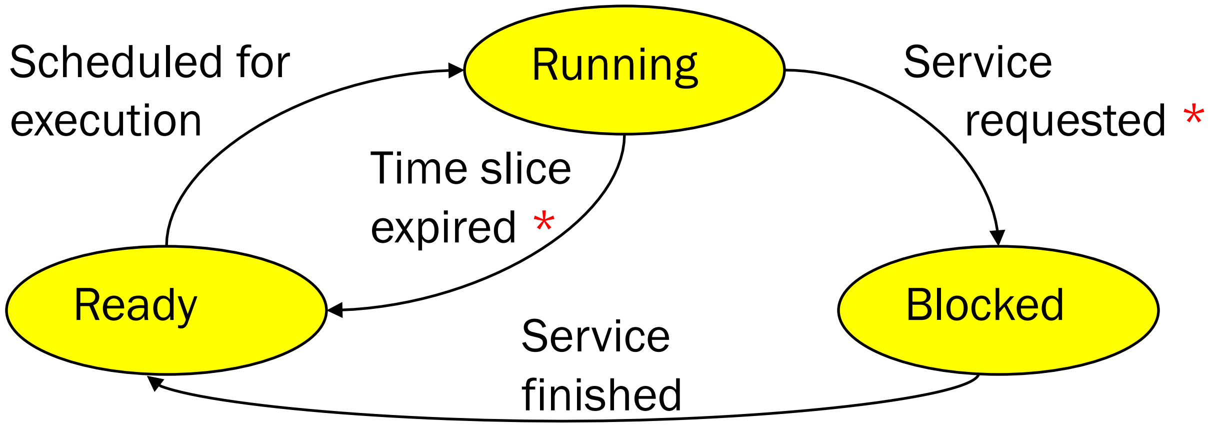
Private Control Flow: Reality



Multiple processes are time-sliced to run concurrently

OS occasionally preempts running process to give other processes their fair share of CPU time

Process Status Transitions



* Preempting transition

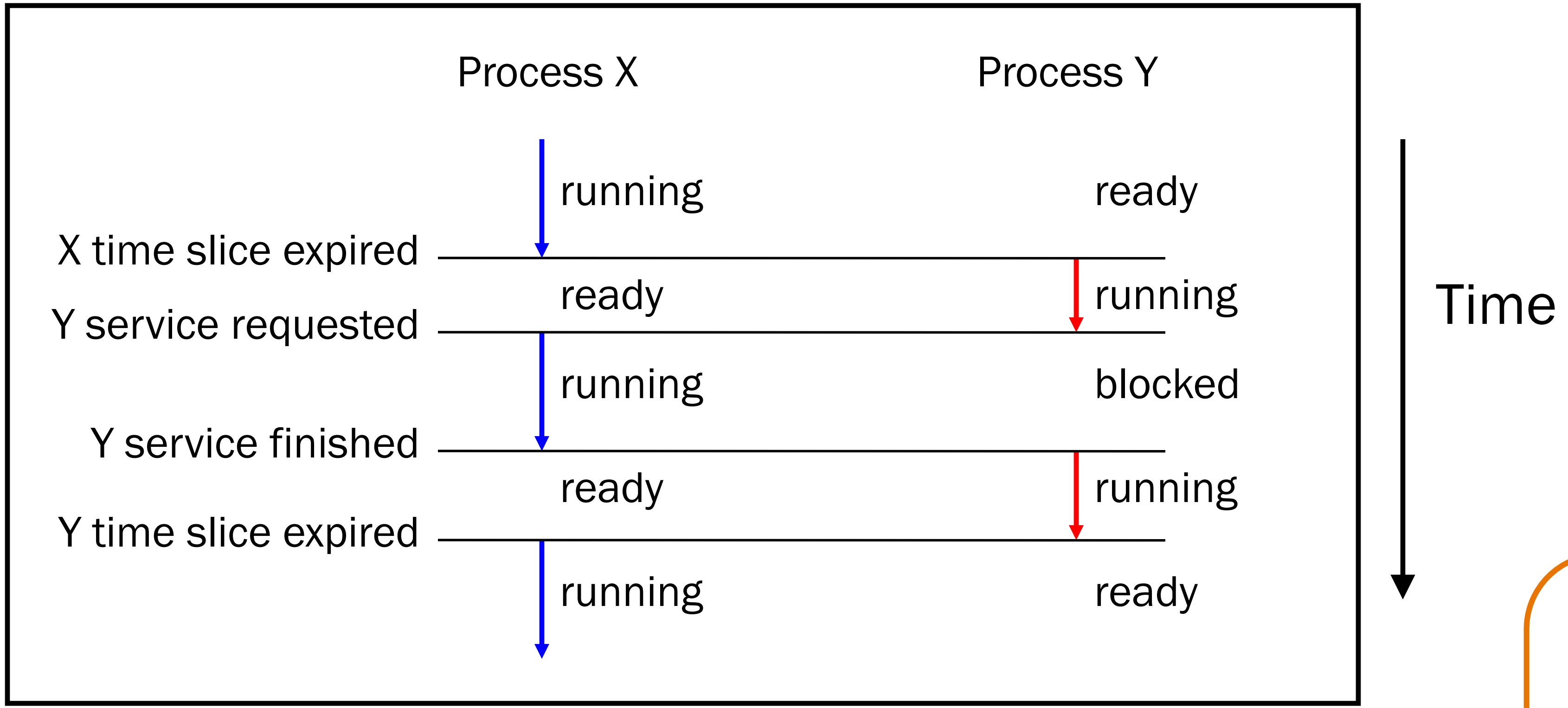
Scheduled for execution: OS selects some process from ready set and assigns CPU to it

Time slice expired: OS moves running process to ready set because process consumed its fair share of CPU time

Service requested: OS moves running process to blocked set because it requested a (time consuming) system service (often I/O)

Service finished: OS moves blocked process to ready set because the requested service finished

Process Status Transitions Over Time

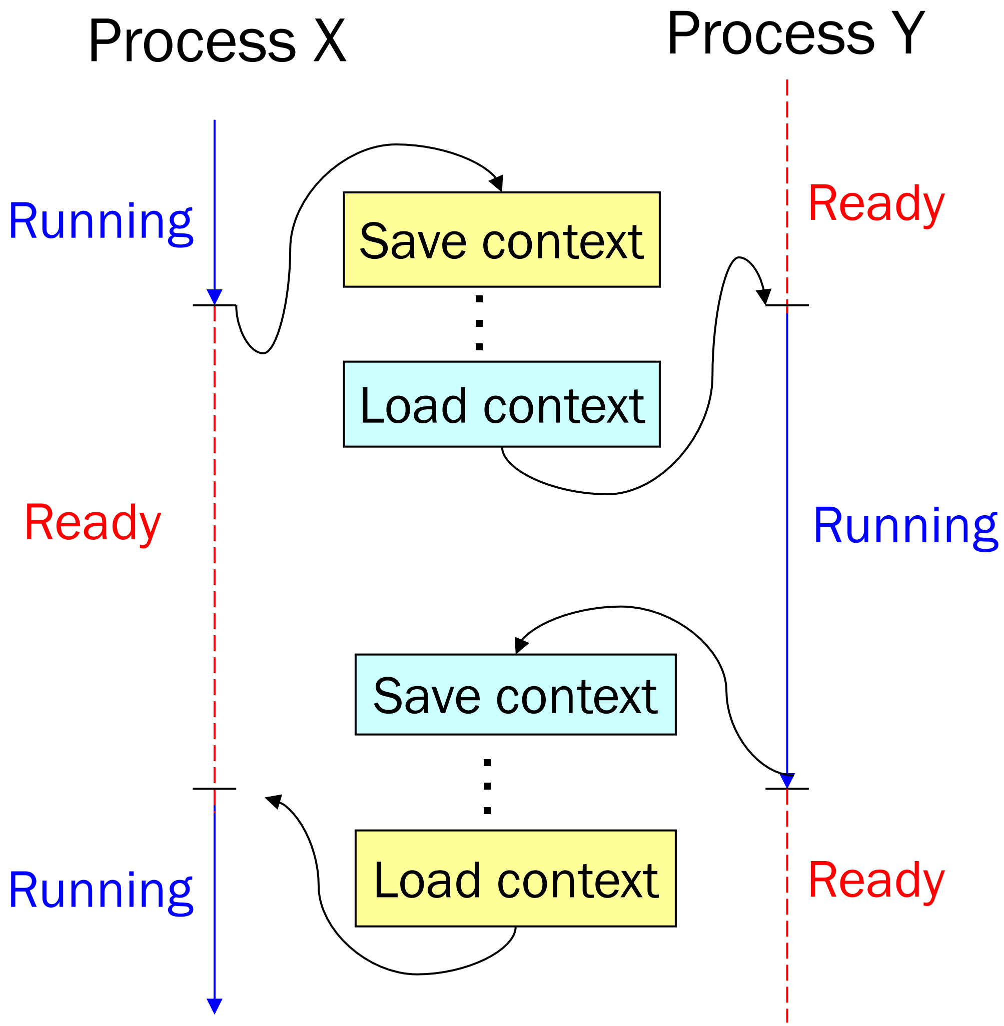


	Process X	Process Y
	running	ready
X time slice expired		
	ready	running
Y service requested		
	running	blocked
Y service finished		
	ready	running
Y time slice expired		
	running	ready

Time

Throughout its lifetime a process’s status switches between running, ready, and blocked

Context Switch



Context switch:

- OS saves context of running process
- OS loads context of some ready process
- OS passes control to newly restored process

Agenda

Processes

- Illusion: Private address space
- Illusion: Private control flow

Process management in C

- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control

- Signals
- Alarms

System-Level Process Management Functions

Function	Description
exit()	Terminate the process
fork()	Create a child process
wait()	Wait for child process termination
execvp()	Execute a program in current process

Why Create New Processes?

Why create a new process?

- Scenario 1: Program wants to run an additional instance of itself
  - E.g., **web server** receives request; creates additional instance of itself to handle the request; original instance continues listening for requests
- Scenario 2: Program wants to run a different program
  - E.g., **shell** receives a command; creates an additional instance of itself; additional instance overwrites itself with requested program to handle command; original instance continues listening for commands

How to create a new process?

- A "parent" process **forks** a "child" process
- (Optionally) child process overwrites itself with a new program, after performing appropriate setup

`fork` System-Level Function

`pid_t fork(void);`

- Create a new process by duplicating the calling process
- New (child) process is an exact duplicate* of the calling (parent) process
- * Exception: the return value of the call to fork (wait 1 slide)

fork() is called once in parent process

fork() returns twice

- Once in parent process
- Once in child process

`fork` and Return Values

Return value of fork has meaning

- In child, `fork()` returns 0
- In parent, `fork()` returns process id of child

```
pid = fork();
if (pid == 0)
{
   /* in child */
   …
}
else
{
   /* in parent */
   …
}
```

Programs With Processes

Parent process and child process run **concurrently**

- Two CPUs available ⇒
  - Parent process and child process run in **parallel**
- Fewer than two CPUs available ⇒
  - Parent process and child process run **serially**
  - OS provides the **illusion** of parallel execution
    - OS causes context switches between the two processes
    - (Recall ***Exceptions and Processes*** lecture)

Reality: Each ArmLab computer has 96 CPUs

Simplifying assumption: there is only one CPU

- We'll speak of "which process gets **the** CPU"
- But which process gets the CPU first? Unknown!

Simple `fork` Example

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

What is the output?

Simple `fork` Example Trace 1 (1)

Parent prints “one”

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 1 (2)

Parent forks child

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 1 (3)

OS gives CPU to child; child prints “two”

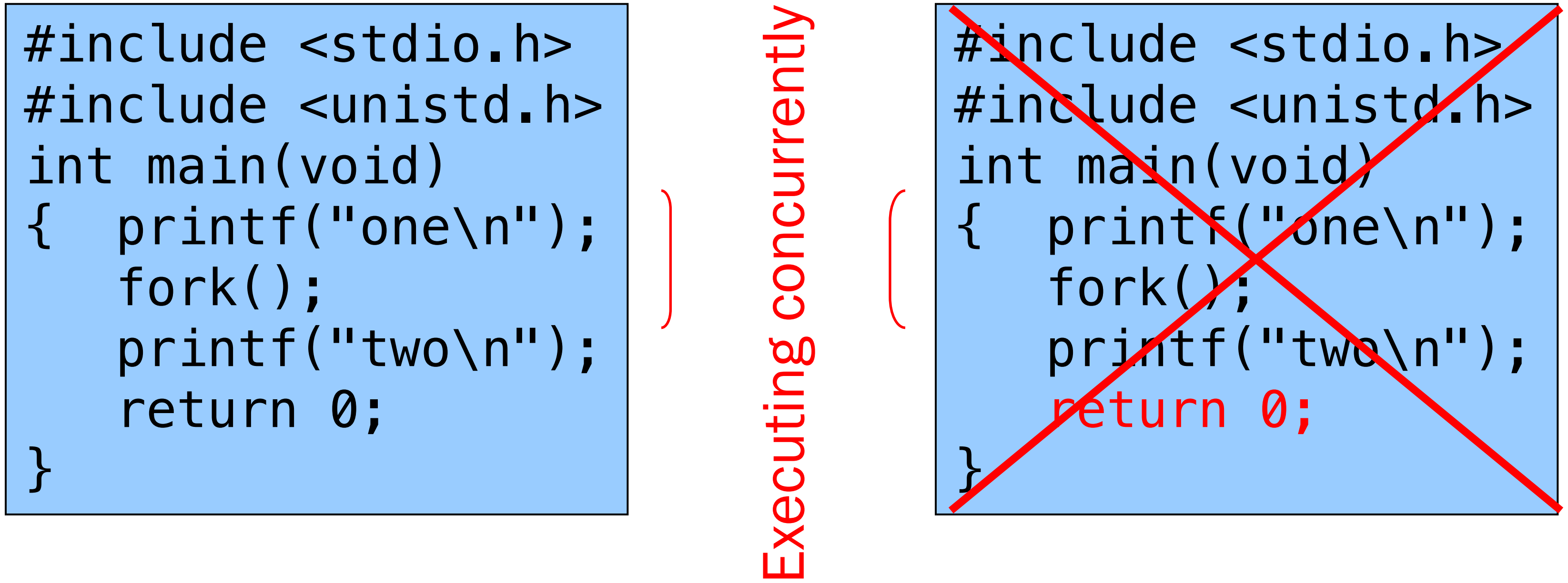
```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 1 (4)

Child exits



```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

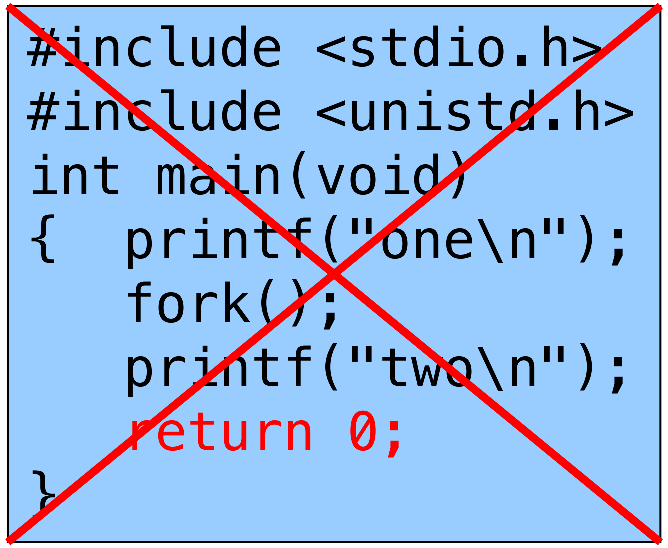
Simple `fork` Example Trace 1 (5)

OS gives CPU to parent; parent prints “two”

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 1 (6)

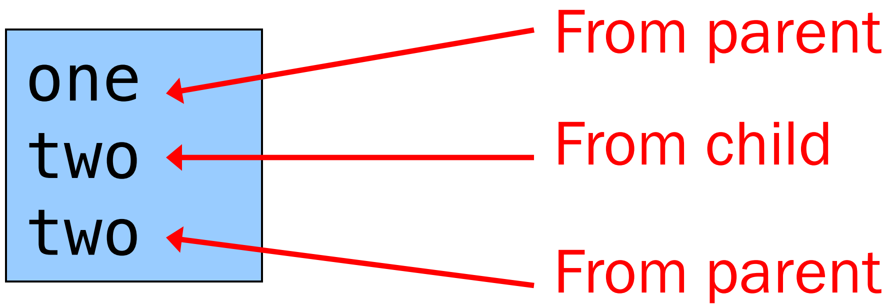
OS gives CPU to parent; parent prints “two”



```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 1 Output

Output:



```
one
two
two
```

From parent

From child

From parent

Simple `fork` Example Trace 2 (1)

Parent prints “one”

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 2 (2)

Parent forks child

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 2 (3)

OS gives CPU to parent; parent prints “two”

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 2 (4)

Parent exits

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 2 (5)

OS gives CPU to child; child prints "two"

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

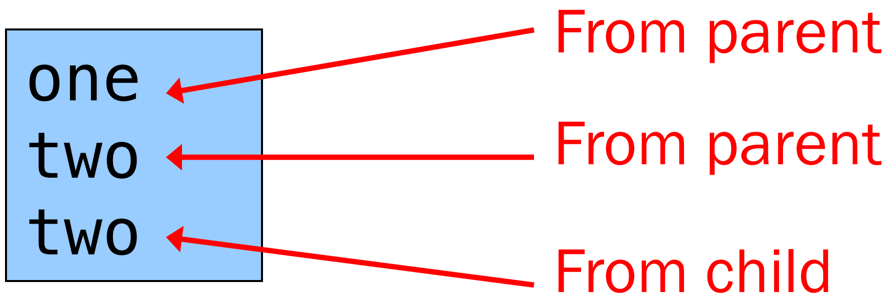
Simple `fork` Example Trace 2 (6)

Child exits

```
#include <stdio.h>
#include <unistd.h>
int main(void)
{  printf("one\n");
   fork();
   printf("two\n");
   return 0;
}
```

Simple `fork` Example Trace 2 Output

Output:



```
one
two
two
```

From parent

From parent

From child

iClicker Question

Q: What is the output of this program?

A. child: 0
   parent: 2

B. parent: 2
   child: 0

C. child: 0
   parent: 1

D. parent: 2
   child: 1

E. A or B

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{   pid_t pid;
    int x = 1;

    pid = fork();
    if (pid == 0)
    {   x--;
        printf("child: %d\n", x);
        exit(0);
    }
    else
    {   x++;
        printf("parent: %d\n", x);
        exit(0);
    }
}
```

The answer is E.

See following slides.

`fork` Example Trace 1 (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 1 (2)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

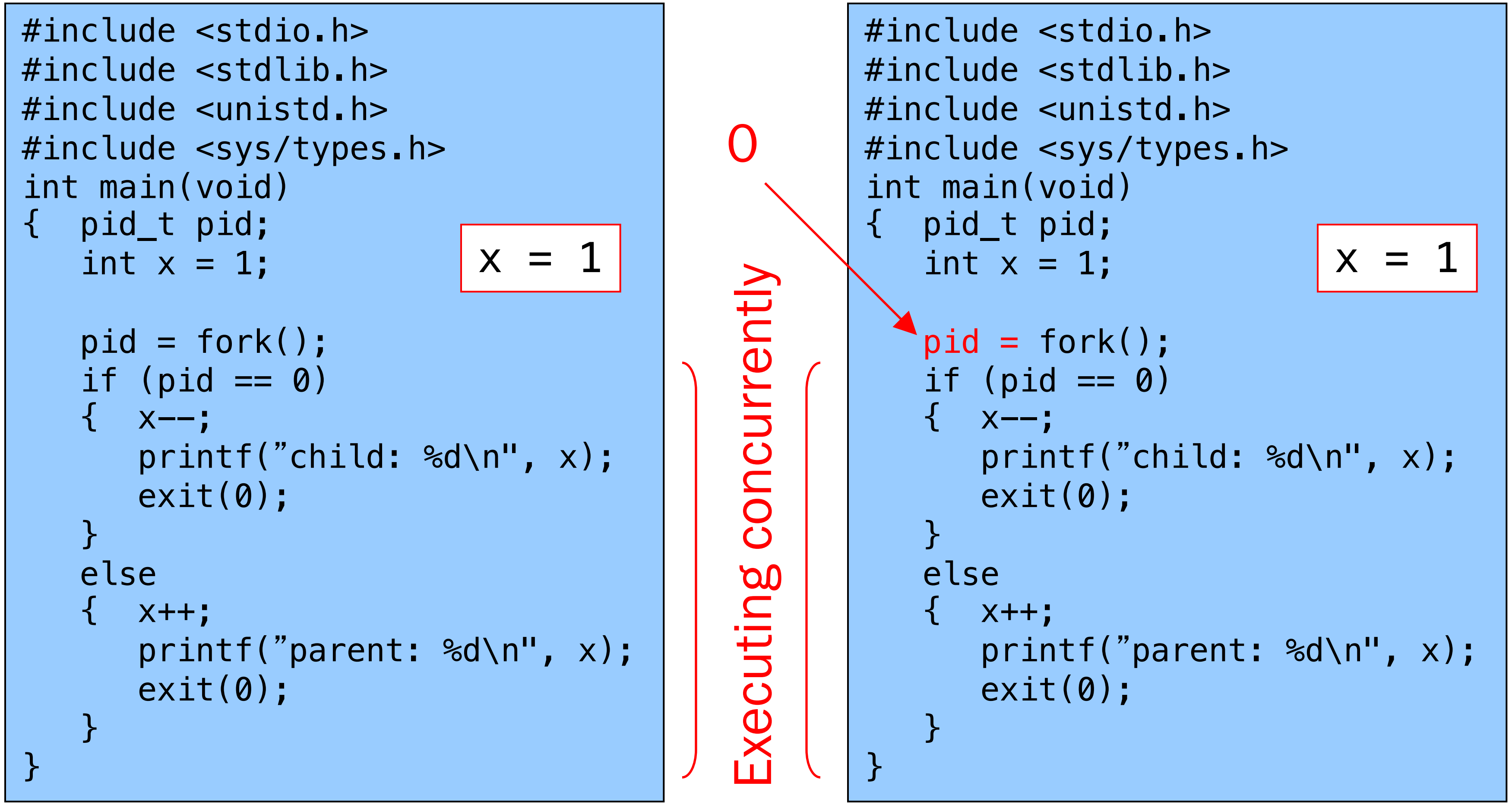
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 1 (3)

Assume OS gives CPU to child



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

0

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 1 (4)

Child decrements its x, and prints “child: 0”

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 0

`fork` Example Trace 1 (5)

Child exits; OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 0

`fork` Example Trace 1 (6)

In parent, fork() returns process id of child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Process id of child

`fork` Example Trace 1 (7)

Parent increments its x, and prints “parent: 2”

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

`fork` Example Trace 1 (8)

Parent exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

`fork` Example Trace 1 Output

Example trace 1 output:

```
Child: 0
Parent: 2
```

`fork` Example Trace 2 (1)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (2)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (3)

Assume OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

Process ID of child

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (4)

Parent increments its x and prints "parent: 2"

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

Executing concurrently

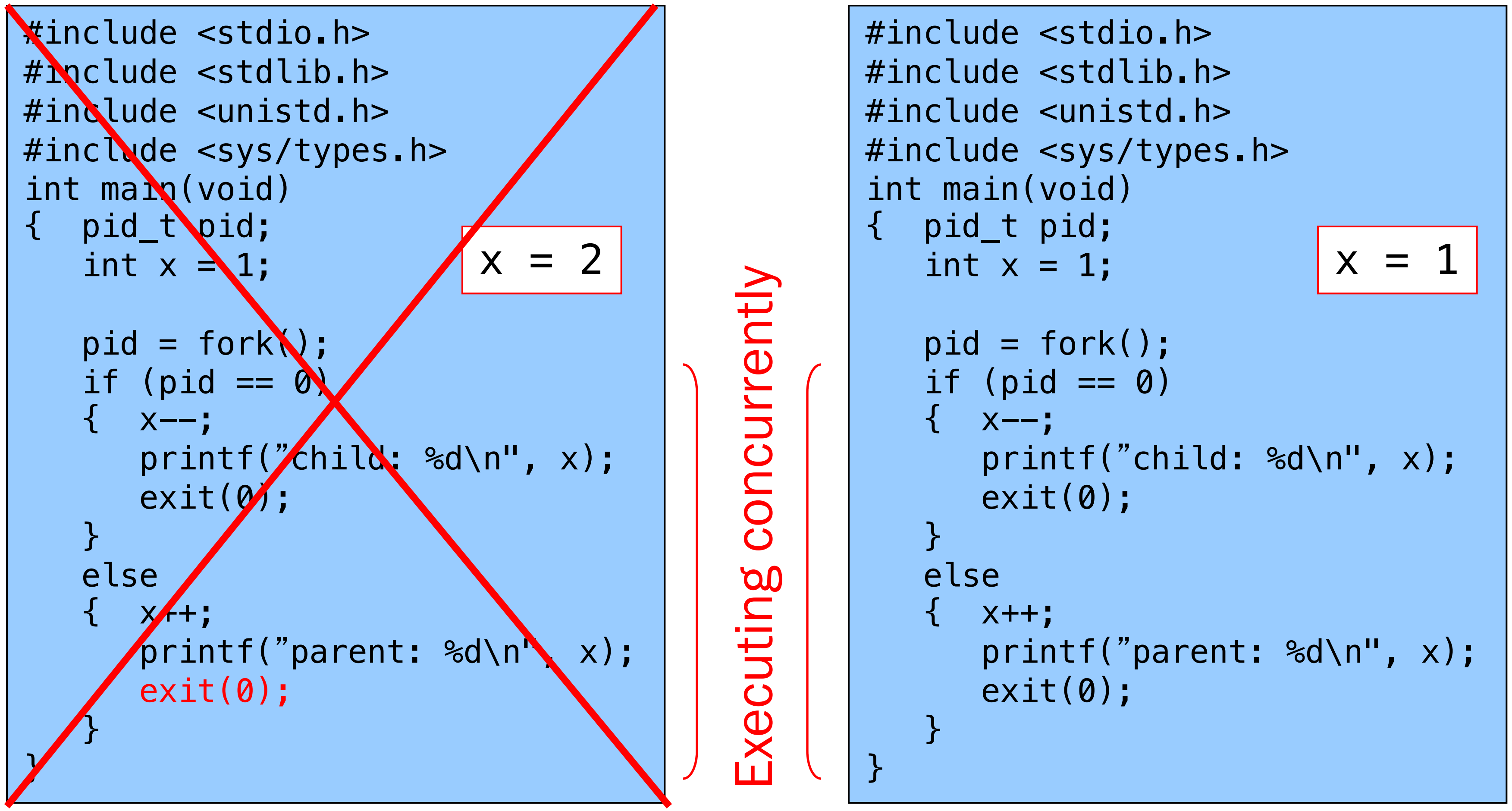
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (5)

Parent exits; OS gives CPU to child



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 2

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (6)

In child, fork() returns 0

0

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 1

`fork` Example Trace 2 (7)

Child decrements its x and prints “child: 0”

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

```
x = 0
```

`fork` Example Trace 2 (8)

Child exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf(”child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf(”parent: %d\n", x);
      exit(0);
   }
}
```

x = 0

`fork` Example Trace 2 Output

Example trace 2 output:

```
Parent: 2
Child: 0
```

```
armlab01:~/Test$ for i in `seq 1 10000`; do ./fpe | head –
n 1; done | sort | uniq –c
     56 child: 0
   9944 parent: 2
```

iClicker Question

Q: Must we do exit(0) instead of return 0; here?

A. Yes, the program will not work with return statements

B. No, but it’s good programming practice for forking programs

C. No, but we need to in some other forking programs

D. No, this is actually a bug in this program and should be return 0 instead

```c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
int main(void)
{  pid_t pid;
   int x = 1;

   pid = fork();
   if (pid == 0)
   {  x--;
      printf("child: %d\n", x);
      exit(0);
   }
   else
   {  x++;
      printf("parent: %d\n", x);
      exit(0);
   }
}
```

B or C

(Consider if this fork were in a deeply nested function stack, not main.)

`wait` System-Level Function

Problem:

- How to control execution order?

Solution:

- Parent calls `wait()`

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

- Suspends execution of the calling process until one of its children terminates
- If status is not NULL, stores status information in the int to which it points; this integer can be inspected with macros [see man page for details].
- On success, returns the process ID of the terminated child
- On error, returns -1
- (a child that has exited is a "zombie" until parent does the `wait()`, so the parent should harvest (or reap) its children... more later)

Paraphrasing man page

iClicker Question

Q: What is the output of this program?

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

A. child
   parent

B. parent
   child

C. something other than A or B

D. A or B

E. A or C

The answer is A.

See following slides.

`wait` Example Trace 1 (1)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 (2)

OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 (3)

Parent calls `wait()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 (4)

OS gives CPU to child

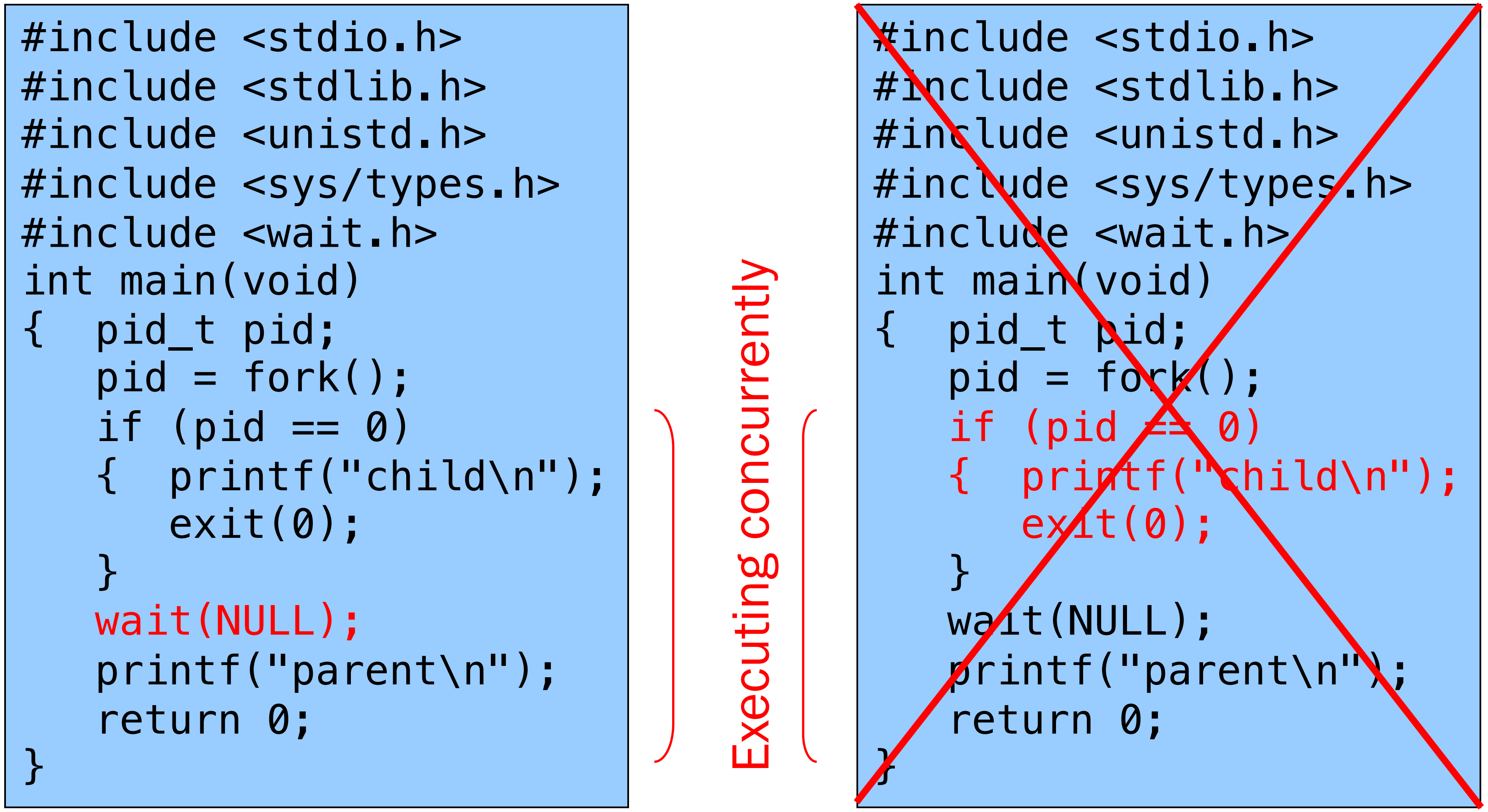
```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 (5)

Child prints “child” and exits



```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 (6)

Parent returns from call of wait(), prints “parent”, exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 1 Output

Example trace 1 output

```
child
parent
```

`wait` Example Trace 2 (1)

Parent forks child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 2 (2)

OS gives CPU to child

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 2 (3)

Child prints “child” and exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{   pid_t pid;
    pid = fork();
    if (pid == 0)
    {   printf("child\n");
        exit(0);
    }
    wait(NULL);
    printf("parent\n");
    return 0;
}
```

Executing concurrently

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{   pid_t pid;
    pid = fork();
    if (pid == 0)
    {   printf("child\n");
        exit(0);
    }
    wait(NULL);
    printf("parent\n");
    return 0;
}
```

`wait` Example Trace 2 (4)

OS gives CPU to parent

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 2 (5)

Parent calls `wait()`; returns immediately

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 2 (6)

Parent prints “parent” and exits

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <wait.h>
int main(void)
{  pid_t pid;
   pid = fork();
   if (pid == 0)
   {  printf("child\n");
      exit(0);
   }
   wait(NULL);
   printf("parent\n");
   return 0;
}
```

`wait` Example Trace 2 Output

Example trace 2 output

```
child
parent
```

Same as trace 1 output!

Aside: Orphans and Zombies

Question:

- What happens if parent process does not wait for (reap/harvest) child process?

Answer 1:

- In shell, could cause sequencing problems
- E.g., parent process running shell writes prompt for next command before current command is finished executing

Answer 2:

- In general, child process becomes **zombie** and/or **orphan**

Aside: Orphans and Zombies

Orphan

- A process that has no parent

Zombie

- A process that has terminated but has not been waited for (reaped)

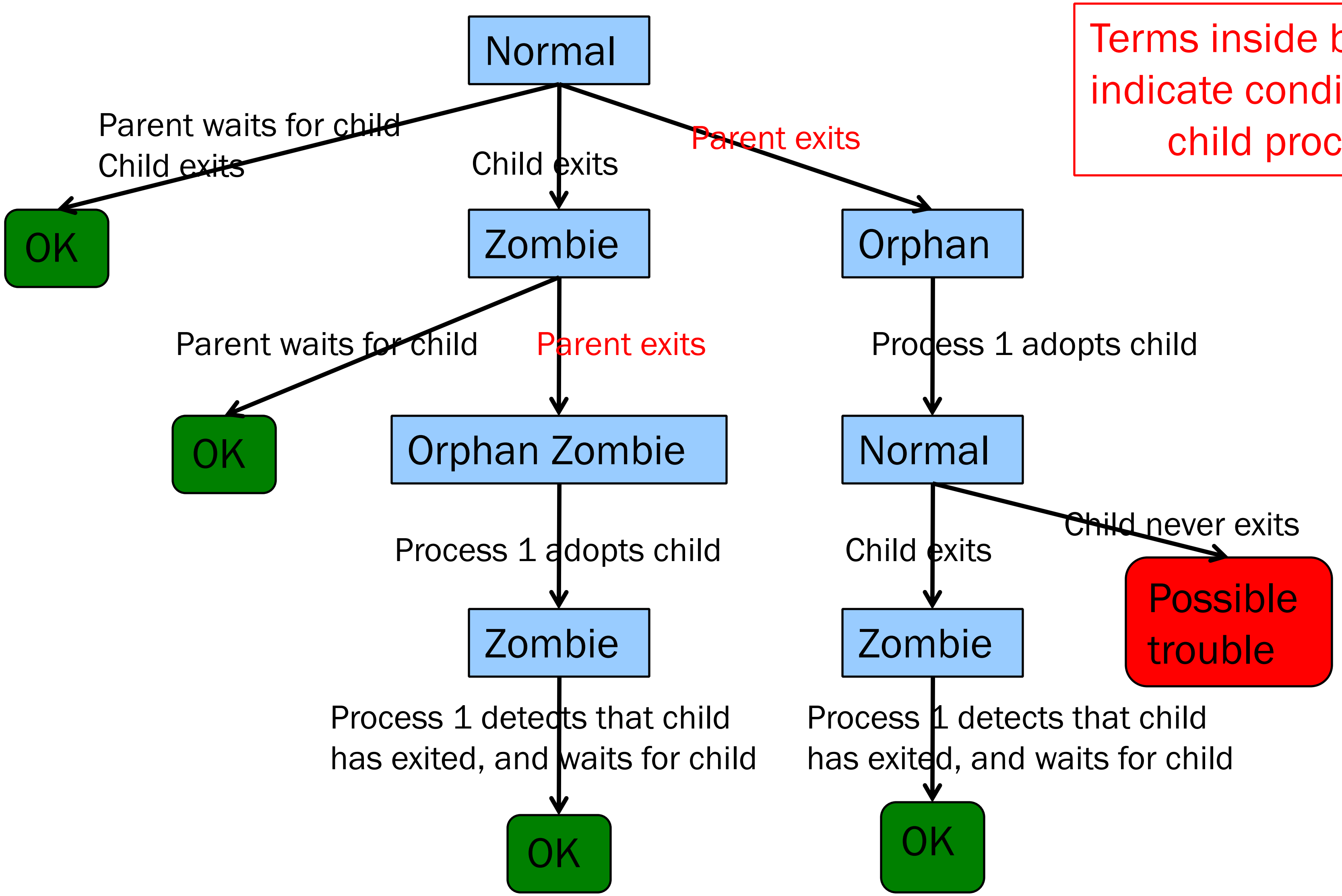
Orphans and zombies

- Clutter Unix data structures unnecessarily
  - OS maintains unnecessary PCBs
- Can become long-running processes

polychlorinated biphenyls?

no, process control blocks: data structures for keeping info about processes

Aside: Orphans and Zombies



Terms inside boxes indicate condition of child process

`execvp` System-Level Function

Problem: How to execute a new program?

- Usually, in the newly-created child process

Solution: `execvp()`

```
int execvp(const char *file, char *const argv[]);
```

- Replaces the current process image with a new process image
- Provides an array of pointers to null-terminated strings that represent the argument list available to the new program
  - The first argument, by convention, should point to the filename associated with the file being executed
  - The array of pointers must be terminated by a NULL pointer

Paraphrasing man page

`execvp` System-Level Function

Example: Execute “cat readme”

```c
char *newCmd;
char *newArgv[3];
newCmd = "cat";
newArgv[0] = "cat";
newArgv[1] = "readme";
newArgv[2] = NULL;
execvp(newCmd, newArgv);
```

- First argument: name of program to be executed
- Second argument: argv to be passed to main() of new program
  - Must begin with program name, end with NULL

`execvp` Failure

```
char *newCmd;
char *newArgv[3];
newCmd = "cat";
newArgv[0] = "cat";
newArgv[1] = "readme";
newArgv[2] = NULL;
execvp(newCmd, newArgv);
fprintf(stderr, "exec failed\n");
exit(EXIT_FAILURE);
```

`fork()`

- If successful, returns **two** times
    - Once in parent
    - Once in child

`execvp()`

- If successful, returns **zero** times
    - Calling program is overwritten with new program
- Corollary:
    - If `execvp()` returns, then it must have failed

execvp Example

```
$ cat readme
This is my
readme file.
```

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{   char *newCmd;
    char *newArgv[3];
    newCmd = "cat";
    newArgv[0] = "cat";
    newArgv[1] = "readme";
    newArgv[2] = NULL;
    execvp(newCmd, newArgv);
    fprintf(stderr, "exec failed\n");
    return EXIT_FAILURE;
}
```

What is the output?

`execvp` Example Trace (1)

Process creates arguments to be passed to `execvp()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{   char *newCmd;
    char *newArgv[3];
    newCmd = "cat";
    newArgv[0] = "cat";
    newArgv[1] = "readme";
    newArgv[2] = NULL;
    execvp(newCmd, newArgv);
    fprintf(stderr, "exec failed\n");
    return EXIT_FAILURE;
}
```

`execvp` Example Trace (2)

Process executes `execvp()`

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int main(void)
{   char *newCmd;
    char *newArgv[3];
    newCmd = "cat";
    newArgv[0] = "cat";
    newArgv[1] = "readme";
    newArgv[2] = NULL;
    execvp(newCmd, newArgv);
    fprintf(stderr, "exec failed\n");
    return EXIT_FAILURE;
}
```

`execvp` Example Trace (3)

`cat` program executes in same process

```
cat program

with argv array:
   argv[0] = "cat"
   argv[1] = "readme"
   argv[2] = NULL
```

`execvp` Example Trace (4)

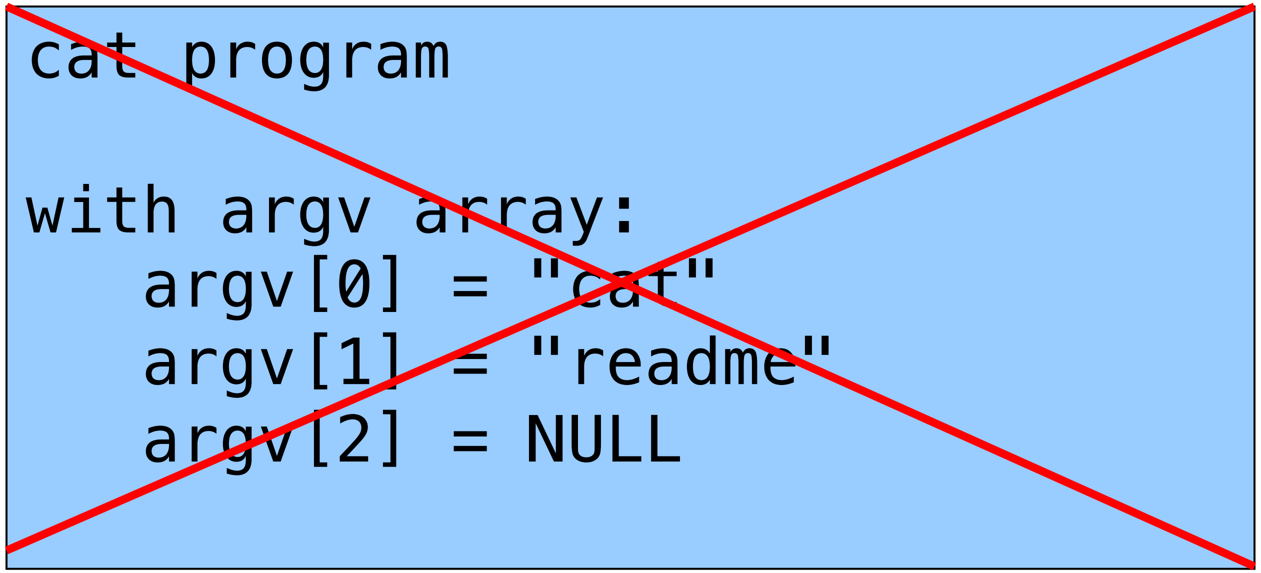
`cat` program writes “This is my\nreadme file.”

```
cat program

with argv array:
   argv[0] = "cat"
   argv[1] = "readme"
   argv[2] = NULL
```

`execvp` Example Trace (5)

`cat` program terminates



```
cat program

with argv array:
    argv[0] = "cat"
    argv[1] = "readme"
    argv[2] = NULL
```

Output

```
This is my
readme file.
```

Aside: `system` Function

Common combination of operations

- `fork()` to create a new child process
- `execvp()` to execute new program in child process
- `wait()` in the parent process for the child to complete

Single call that combines all three

- `int system(const char *cmd);`

Example

```
#include <stdlib.h>
int main(void)
{  system("cat readme");
   return 0;
}
```

Shell Structure

Parent (shell) reads & parses the command line
- E.g., “cat readme”

Parent forks child

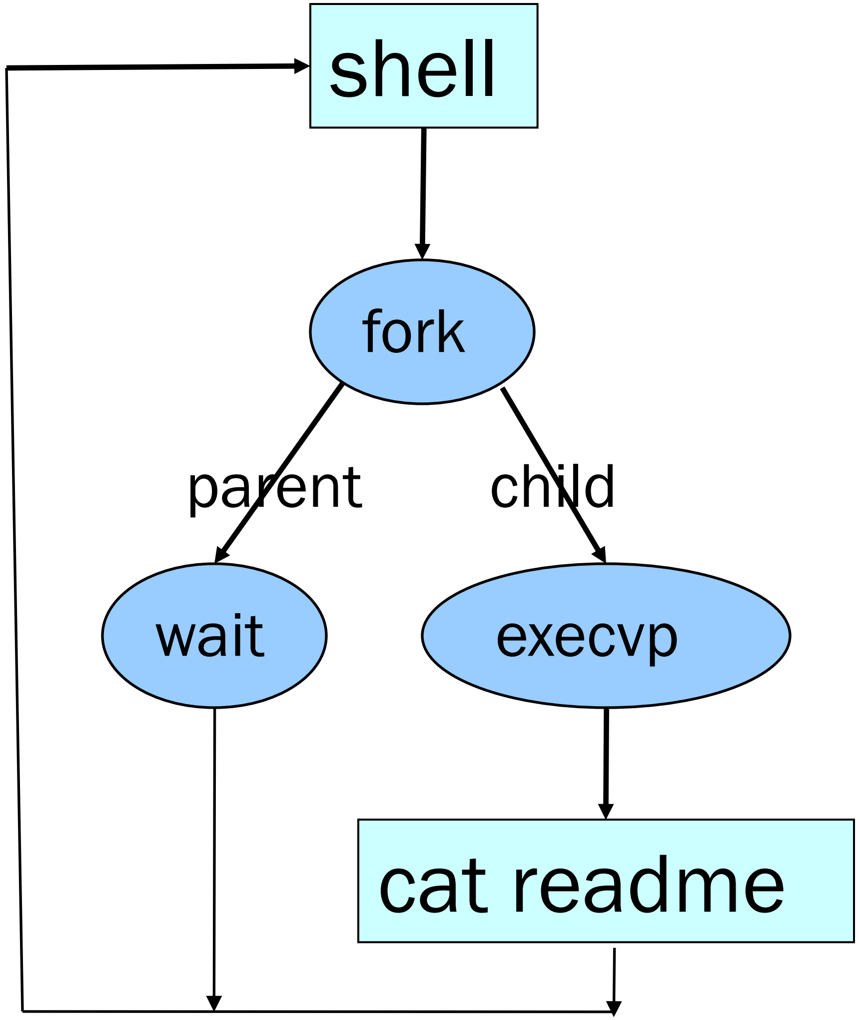
Parent waits

Child calls execvp to execute command

Child exits

Parent returns from wait

Parent repeats



Simple Shell Code

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

Simple Shell Trace (1)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

Parent reads and parses command line

Parent assigns values to `somepgm` and `someargv`

Simple Shell Trace (2)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

`fork()` creates child process

Which process gets the CPU first? Let’s assume the parent…

Simple Shell Trace (3)

Parent Process

Child's pid

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

In parent, pid != 0; parent waits; OS gives CPU to child

Simple Shell Trace (4)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

0

```
Parse command line
Assign values to somefile, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

In child, pid == 0; child calls `execvp()`

Simple Shell Trace (5)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

```
somepgm
With someargv
as argv param
```

In child, somepgm overwrites shell program;
`main()` is called with `someargv` as `argv` parameter

Simple Shell Trace (6)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

executing concurrently

Child Process

~~somepgm~~
~~With someargv~~
~~as argv param~~

Somepgm executes in child, and eventually exits

Simple Shell Trace (7)

Parent Process

```
Parse command line
Assign values to somepgm, someargv
pid = fork();
if (pid == 0) {
   /* in child */
   execvp(somepgm, someargv);
   fprintf(stderr, "exec failed\n");
   exit(EXIT_FAILURE);
}
/* in parent */
wait(NULL);
Repeat the previous
```

Parent returns from `wait()` and repeats

Aside: background processes

Unix shell lets you run a process “in the background”

```
$  compute <my-input >my-output &
```

How it’s implemented in the shell:

Don’t wait() after the fork!

But: must clean up zombie processes

```
waitpid(0, &status, WNOHANG)
```

(more info: “man 2 wait”)

When to do it?

Every time around the main loop, or

When parent receives the SIGCHLD signal.

One or the other,
don’t need both!

Agenda

Processes

- Illusion: Private address space
- Illusion: Private control flow

Process management in C

- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control

- Signals
- Alarms

Process Control Examples

Exactly what happens when you:

Type Ctrl-c?

- Keystroke generates **interrupt**
- OS handles interrupt
- OS sends a 2/SIGINT **signal**

Type Ctrl-z?

- Keystroke generates **interrupt**
- OS handles interrupt
- OS sends a 20/SIGTSTP **signal**

Sending Signals via Keystrokes

User can send three signals from keyboard:

- Ctrl-c ⇒ 2/SIGINT signal
  - Default action is "terminate"
- Ctrl-z ⇒ 20/SIGTSTP signal
  - Default action is "stop until next 18/SIGCONT"
- Ctrl-\ ⇒ 3/SIGQUIT signal
  - Default action is "terminate"

Examples of Non-keyboard Signals

Process makes illegal memory reference

- Segmentation fault occurs
- OS gains control of CPU
- OS sends 11/SIGSEGV signal to process
- Process receives 11/SIGSEGV signal
- Default action for 11/SIGSEGV signal is “terminate”

https://xkcd.com/371/

Signals Overview

Signal: A notification of an exception

Typical signal sequence:

- Process P is executing
- Exception occurs (interrupt, trap, fault, or abort)
- OS gains control of CPU
- OS wishes to inform process P that something happened
- OS **sends** a signal to process P
  - OS sets a bit in `pending` **bit vector** of process P
  - Indicates that OS is sending a signal of type X to process P
  - A signal of type X is **pending** for process P

Signals Overview (cont.)

Typical signal sequence (cont.):

- Sometime later…
- OS is ready to give CPU back to process P
- OS checks `pending` for process P, sees that signal of type X is pending
- OS forces process P to **receive** signal of type X
    - OS clears bit in process P's `pending`
- Process P executes action for signal of type X
    - Normally process P executes **default action** for that signal
    - If **signal handler** was installed for signal of type X, then process P executes signal handler
    - Action might terminate process P; otherwise…
- Process P resumes where it left off

Sending Signals via Commands

User can send any signal by executing command:

`kill` command

- `kill –sig pid`
- Send a signal of type `sig` to process `pid`
- No `–sig` option specified ⇒ sends 15/SIGTERM signal
  - Default action for 15/SIGTERM is “terminate”
- You must own process `pid` (or have admin privileges)
- Commentary: Better command name would be `sendsig`

Examples

- `kill –2 1234`
- `kill –SIGINT 1234`
  - Same as pressing Ctrl-c if process 1234 is running in foreground
- `kill –2 %1`
  - Same as above, if process 1234 is running as background job 1

Process Control Implementation (cont.)

Exactly what happens when you:

Issue a `kill –sig pid` command?

- `kill` command executes **trap**
- OS handles trap
- OS sends a `sig` **signal** to the process whose id is `pid`

Issue a `fg` or `bg` command?

- `fg` or `bg` command executes **trap**
- OS handles trap
- OS sends a 18/SIGCONT **signal** (and does some other things too!)

Signals signals everywhere

List of the predefined signals, learn many details with these commands:

```
$ kill -l
 1) SIGHUP       2) SIGINT       3) SIGQUIT      4) SIGILL
 5) SIGTRAP      6) SIGABRT      7) SIGBUS       8) SIGFPE
 9) SIGKILL     10) SIGUSR1     11) SIGSEGV     12) SIGUSR2
13) SIGPIPE     14) SIGALRM     15) SIGTERM     17) SIGCHLD
18) SIGCONT     19) SIGSTOP     20) SIGTSTP     21) SIGTTIN
22) SIGTTOU     23) SIGURG      24) SIGXCPU     25) SIGXFSZ
26) SIGVTALRM   27) SIGPROF     28) SIGWINCH    29) SIGIO
30) SIGPWR      31) SIGSYS      34) SIGRTMIN    35) SIGRTMIN+1
36) SIGRTMIN+2  37) SIGRTMIN+3  38) SIGRTMIN+4  39) SIGRTMIN+5
40) SIGRTMIN+6  41) SIGRTMIN+7  42) SIGRTMIN+8  43) SIGRTMIN+9
44) SIGRTMIN+10 45) SIGRTMIN+11 46) SIGRTMIN+12 47) SIGRTMIN+13
48) SIGRTMIN+14 49) SIGRTMIN+15 50) SIGRTMAX-14 51) SIGRTMAX-13
52) SIGRTMAX-12 53) SIGRTMAX-11 54) SIGRTMAX-10 55) SIGRTMAX-9
56) SIGRTMAX-8  57) SIGRTMAX-7  58) SIGRTMAX-6  59) SIGRTMAX-5
60) SIGRTMAX-4  61) SIGRTMAX-3  62) SIGRTMAX-2  63) SIGRTMAX-1
64) SIGRTMAX

$ man 7 signal
```

See Bryant & O’Hallaron book for more actions, triggering exceptions,
and how the application program can define signals with unused values

Sending Signals via Function Calls

Program can send any signal by calling function:

`raise()` function

- `int raise(int iSig);`
- Commands OS to send a signal of type `iSig` to calling process
- Returns 0 to indicate success, non-0 to indicate failure

Example:

- `iRet = raise(SIGINT);`
  - Send a 2/SIGINT signal to calling process

One clever use case:

https://www.gnu.org/software/libc/manual/html_node/Signaling-Yourself.html

Sending Signals via Function Calls

`kill()` function

- `int kill(pid_t iPid, int iSig);`
- Sends a `iSig` signal to the process `iPid`
- Equivalent to `raise(iSig)` when `iPid` is the id of current process
- You must own process `pid` (or have admin privileges)
- Commentary: Better function name would be `sendsig()`

Example

- `iRet = kill(1234, SIGINT);`
  - Send a 2/SIGINT signal to process 1234

Handling Signals

Each signal type has a default action

- For most signal types, default action is “terminate”

A program can **install** a **signal handler**

- To change action of (almost) any signal type

Installing a Signal Handler

`signal()` function

- `sighandler_t signal(int iSig, sighandler_t pfHandler);`

- Install function `pfHandler` as the handler for signals of type `iSig`
- `pfHandler` is a function pointer:

  `typedef void (*sighandler_t)(int);`
- Return the old handler on success, `SIG_ERR` on error
- After call, `(*pfHandler)` is invoked whenever process receives a signal of type `iSig`

SIG_DFL

Predefined value: **SIG_DFL**

Use as argument to `signal()` to **restore default action**

```
int main(void)
{   …
    signal(SIGINT, somehandler);
    …
    signal(SIGINT, SIG_DFL);
    …
}
```

Subsequently, process will handle 2/SIGINT signals using default action for 2/SIGINT signals ("terminate")

SIG_IGN

Predefined value: **SIG_IGN**

Use as argument to `signal()` to **ignore signals**

```
int main(void)
{  …
   signal(SIGINT, SIG_IGN);
   …
}
```

Subsequently, process will ignore 2/SIGINT signals

Uncatchable Signals

Special cases: A program *cannot* install a signal handler for signals of type:

- 9/SIGKILL
  - Default action is "terminate"
- 19/SIGSTOP
  - Default action is "stop until next 18/SIGCONT"

Signal Handling Example 1

Program testsignal.c:

```c
#define _GNU_SOURCE /* Use modern handling style */
#include <stdio.h>
#include <signal.h>

static void myHandler(int iSig)
{  printf("In myHandler with argument %d\n", iSig);
}

int main(void)
{  signal(SIGINT, myHandler);
   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0; /* Never get here. */
}
```

Error handling code omitted in this and all subsequent programs in this lecture

```
armlab01:~/Test$ ./testsignal
Entering an infinite loop
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^Z
[1]+  Stopped         ./testsignal
armlab01:~/Test$ fg
./signal
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^CIn myHandler with argument 2
^\Quit
```

Signal Handling Example 2

Program testsignalall.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>

static void myHandler(int iSig)
{  printf("In myHandler with argument %d\n", iSig);
}

int main(void)
{  int i;
   /* Install myHandler as the handler
      for all kinds of signals. */
   for (i = 1; i < 65; i++)
      signal(i, myHandler);
   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0; /* Never get here. */
}
```

```
armlab01:~/Test$ ./a.out
signal 9 not handled
signal 19 not handled
signal 32 not handled
signal 33 not handled
Entering an infinite loop
^CIn myHandler with argument 2
^ZIn myHandler with argument 20
^\In myHandler with argument 3
Killed
```

```
armlab01:~$ ps axu | grep 'a.out'
cmoretti 64220  101  0.0   2432 …
armlab01:~$ kill -9 64220
```

Will fail:

```
signal(9, myHandler)
signal(19, myHandler)
...
```

Signal Handling Example 3

Program generates lots of temporary data

- Stores the data in a temporary file
- Must delete the file before exiting

```
…
int main(void)
{  FILE *psFile;
   psFile = fopen("temp.txt", "w");
   …
   fclose(psFile);
   remove("temp.txt");
   return 0;
}
```

Example 3 Problem

What if user types Ctrl-c?

- OS sends a 2/SIGINT signal to the process
- Default action for 2/SIGINT is "terminate"

Problem: The temporary file is not deleted

- Process terminates before `remove("temp.txt")` is executed

Challenge: Ctrl-c could happen at any time

- Which line of code will be interrupted???

Solution: Install a signal handler

- Define a "clean up" function to delete the file
- Install the function as a signal handler for 2/SIGINT

Example 3 Solution

```
…
static FILE *psFile; /* Must be global. */
static void cleanup(int iSig)
{  fclose(psFile);
   remove("temp.txt");
   exit(0);
}
int main(void)
{  …
   psFile = fopen("temp.txt", "w");
   signal(SIGINT, cleanup);
   …
   cleanup(0);  /* or raise(SIGINT); */
   return 0;  /* Never get here. */
}
```

Alarms

`alarm()` function

- `unsigned int alarm(unsigned int uiSec);`
- Send 14/SIGALRM signal after `uiSec` seconds
- Cancel pending alarm if `uiSec` is 0
- Use **wall-clock time**
  - Time spent executing other processes counts
  - Time spent waiting for user input counts
- Return value is irrelevant for our purposes

Used to implement time-outs

Alarm Example 1

Program testalarm.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{  printf("In myHandler with argument %d\n", iSig);
   alarm(2); /* Set another alarm */
}

int main(void)
{  signal(SIGALRM, myHandler);
   alarm(2); /* Set an alarm. */
   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0; /* Never get here. */
}
```

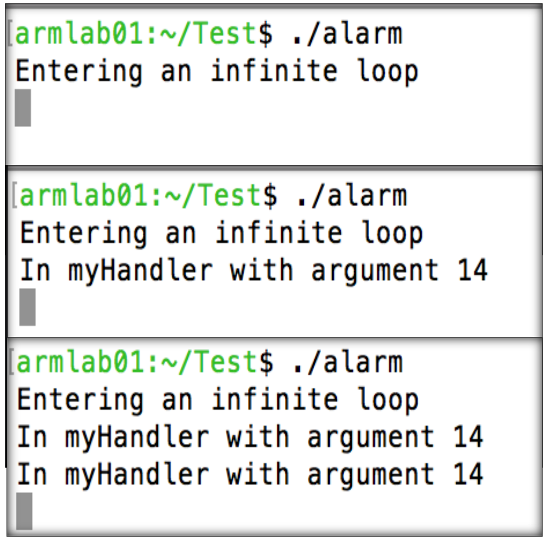
Alarm Example 1

Program testalarm.c:

```c
#define _GNU_SOURCE
#include <stdio.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{  printf("In myHandler with argument %d\n", iSig);
   alarm(2); /* Set another alarm */
}

int main(void)
{  signal(SIGALRM, myHandler);
   alarm(2); /* Set an alarm. */
   printf("Entering an infinite loop\n");
   for (;;)
      ;
   return 0; /* Never get here. */
}
```



```
armlab01:~/Test$ ./alarm
Entering an infinite loop
```

```
armlab01:~/Test$ ./alarm
Entering an infinite loop
In myHandler with argument 14
```

```
armlab01:~/Test$ ./alarm
Entering an infinite loop
In myHandler with argument 14
In myHandler with argument 14
```

Alarm Example 2

Program testalarmtimeout.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{  printf("\nSorry.  You took too long.\n");
   exit(EXIT_FAILURE);
}

int main(void)
{  int i;
   signal(SIGALRM, myHandler);
   printf("Enter a number:  ");
   alarm(5);
   scanf("%d", &i);
   alarm(0);
   printf("You entered the number %d.\n", i);
   return 0;
}
```

Alarm Example 2

Program testalarmtimeout.c:

```
#define _GNU_SOURCE
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <unistd.h>

static void myHandler(int iSig)
{  printf("\nSorry.  You took too long.\n");
   exit(EXIT_FAILURE);
}

int main(void)
{  int i;
   signal(SIGALRM, myHandler);
   printf("Enter a number:  ");
   alarm(5);
   scanf("%d", &i);
   alarm(0);
   printf("You entered the number %d.\n", i);
   return 0;
}
```

```
armlab01:~/Test$ echo 5 |
> ./a.out
Enter a number:
You entered the number 5.

armlab01:~/Test$ (sleep 10;
> echo 5) |
> ./a.out
Enter a number:
Sorry.  You took too long.
```

Agenda

Processes

- Illusion: Private address space
- Illusion: Private control flow

Process management in C

- Creating new processes
- Waiting for termination
- Executing new programs

Unix Process Control

- Signals
- Alarms