Announcements

- CSSS election
  - more information at http://csss.usask.ca
- Computer Science Internship
  - Information session 12:30 - 13:30 on Sept 21
  - https://www.cs.usask.ca/news-and-events/2016/cspip-information-session.php
- Ladies Learning Code is hosting its fourth annual National Learn to Code Day
  - Saturday, September 24
  - http://ladieslearningcode.com/codeday/2016/

Announcements

- Lab 1 solution

Quotes of the Day

- There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. And the other way is to make it so complicated that there are no obvious deficiencies.
  - C.A.R. Hoare
- Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.
  - Martin Golding

Where we finished last class:

Redirection Revisited

- earlier, saw examples like
  `cat < source > destination`
- in `bash` can explicitly redirect file descriptors by preceding the '>' or '<' with the file descriptor number
  - e.g. above equivalent to
    `cat 0< source 1> destination`

    < redirect input **> redirect output** **assuming youre redirecting std out**
- saw `2> file` for redirection of *stderr* earlier

Redirection Revisited

- can also duplicate a file descriptor
  - two file descriptors will refer to the same file
  - full semantics beyond the scope of this course
- in `bash`, accomplished by adding '&' to redirection operators
  - e.g. `m>&n` instead of >

think of file descriptors to be pointers, you get a duplicate of one of the pointers

Redirection Revisited

- common use: to redirect both stdout and stderr to a single file
  - e.g. in `bash`

```
    prog >log 2>&1
```

- evaluation is left-to-right
- note: order of evaluation is important!

```
  prog 2>&1 >log
```

  does something different.

Redirection Revisited

- common use: to redirect both stdout and stderr to a single file
  - e.g. in `bash`

```
    prog >log 2>&1
```

this is the right way

- evaluation is left-to-right
- note: don't open the same file with different file descriptors

```
  prog >log 2>log
```

does something different.

On to new material …

What is a Process?

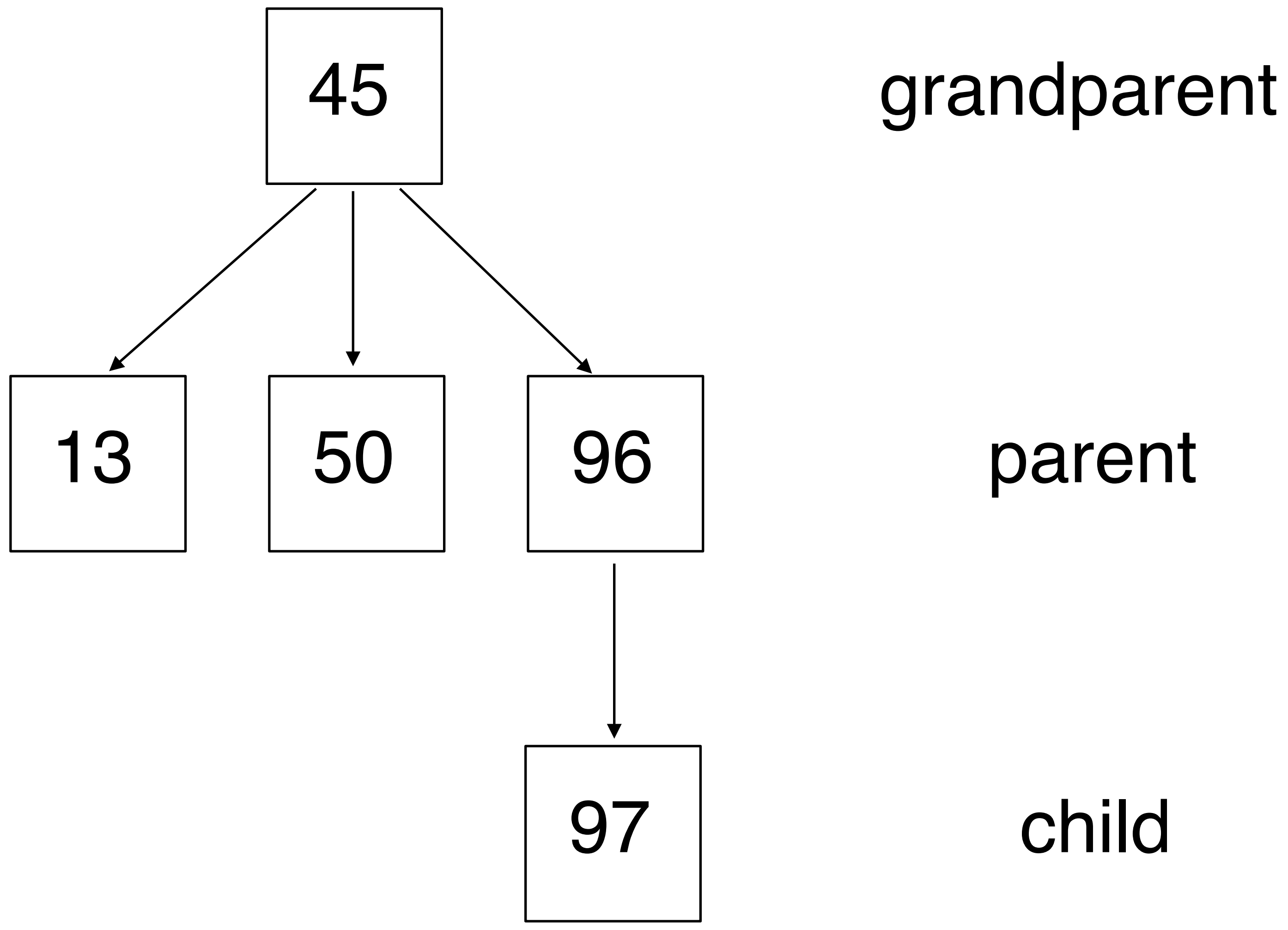
- one definition of a *process*
  - a thread of control in an address space
- recall:
  - a program my invoke several processes
  - a single process can run multiple programs

Basic Process Abstraction in UNIX

- processes exist in a hierarchy
- parent/child/sibling model
  - each process has a unique parent
  - processes can have multiple children
    - each child will be a sibling of the other children
- each process identified by a unique identifier, its *PID* **process identifier**

Basic Process Abstraction in UNIX

- example

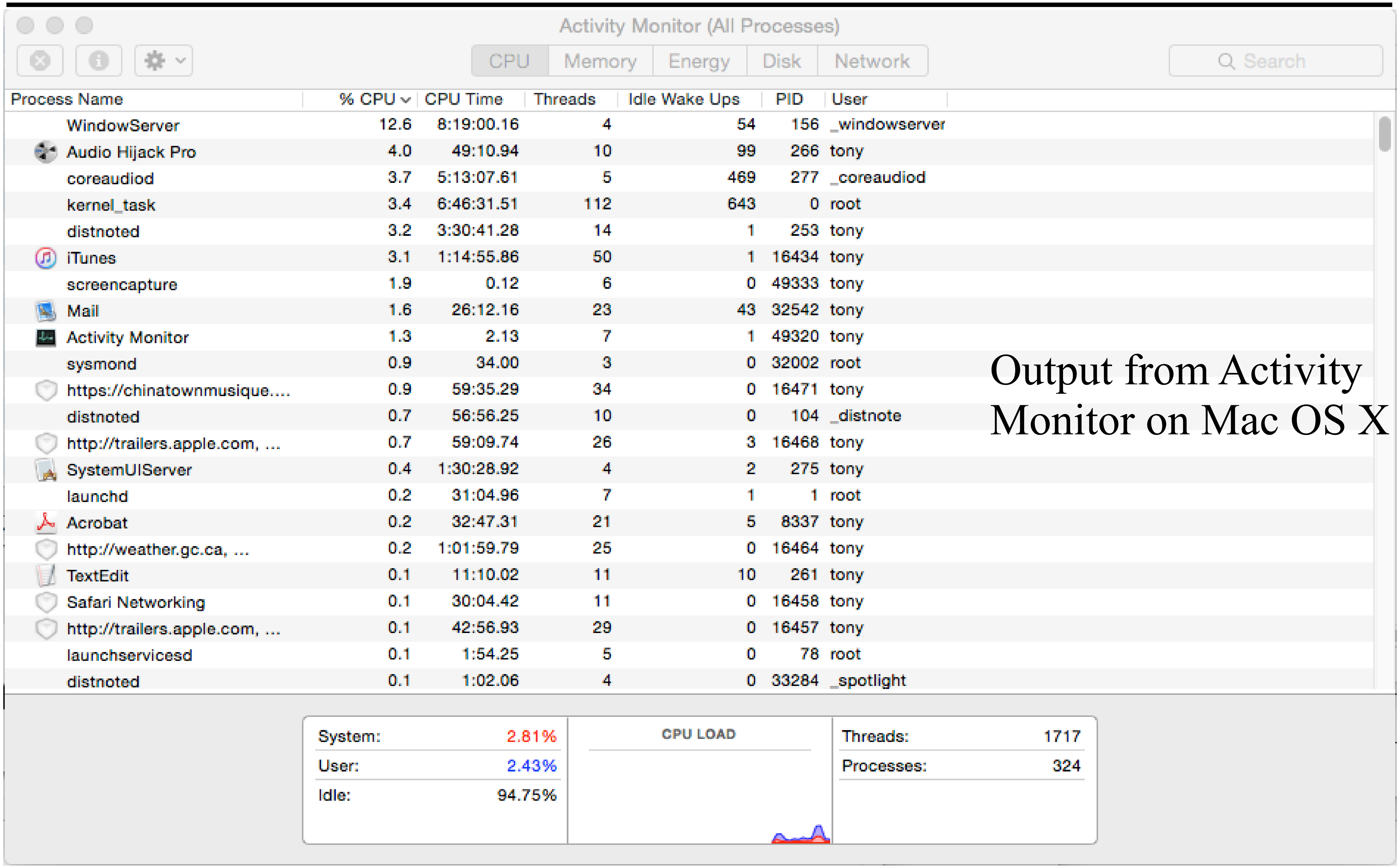


Basic Process Abstraction in UNIX

- abstraction in other operating systems is similar

```
COMMAND        PID        USER    TIME  %KER  %USE  PRI     RSS    SWAP   %MEM  THRD   %CPU
top           2620 administra    0:00   100     0    8    2076     676   0.20     1  40.00
lsass          672 SYSTEM        1h42    28    71    9   80240   77764   7.66    56   0.20
mstsc         2128 administra   25:12    48    51    8    5928    8504   0.57    10   0.12
cmd           1528 administra    0:00    71    28    8    1512    1424   0.14     1   0.05
services       660 SYSTEM        6:56    46    53    9  136580    4372  13.03    20   0.01
dns           1976 SYSTEM        5:24    53    46    8    7428    9064   0.71    14   0.01
mmc           2712 administra    0:08    62    37    8   16464    9108   1.57     5   0.01
svchost       1340 SYSTEM        4:47    41    58    8   24116   17340   2.30    41   0.01
winlogon      2884 SYSTEM        0:05    16    83   13    6412    6028   0.61    15   0.01
winlogon       600 SYSTEM        3:27    57    42   13    4796    7116   0.46    22   0.01
perl          1644 administra    2:16    24    75    8   15720    9752   1.50     4   0.00
dfssvc        1944 SYSTEM        1:52    49    50    8    4724    1892   0.45    11   0.00
svchost       1180 -             1:32    64    35    8    3652    1340   0.35    10   0.00
explorer      3540 administra    1:26    79    20    8   18172    8588   1.73     8   0.00
spoolsv       1720 SYSTEM        1:10    34    65    8    7796    5196   0.74    17   0.00
csrss         1520 SYSTEM        0:01    65    34   13    3024    1076   0.29    11   0.00
explorer       424 administra    0:00    69    30    8   10800    6368   1.03    10   0.00
```

Basic Process Abstraction in UNIX



Process Name	% CPU	CPU Time	Threads	Idle Wake Ups	PID	User
WindowServer	12.6	8:19:00.16	4	54	156	_windowserver
Audio Hijack Pro	4.0	49:10.94	10	99	266	tony
coreaudiod	3.7	5:13:07.61	5	469	277	_coreaudiod
kernel_task	3.4	6:46:31.51	112	643	0	root
distnoted	3.2	3:30:41.28	14	1	253	tony
iTunes	3.1	1:14:55.86	50	1	16434	tony
screencapture	1.9	0.12	6	0	49333	tony
Mail	1.6	26:12.16	23	43	32542	tony
Activity Monitor	1.3	2.13	7	1	49320	tony
sysmond	0.9	34.00	3	0	32002	root
https://chinatownmusique....	0.9	59:35.29	34	0	16471	tony
distnoted	0.7	56:56.25	10	0	104	_distnote
http://trailers.apple.com, ...	0.7	59:09.74	26	3	16468	tony
SystemUIServer	0.4	1:30:28.92	4	2	275	tony
launchd	0.2	31:04.96	7	1	1	root
Acrobat	0.2	32:47.31	21	5	8337	tony
http://weather.gc.ca, ...	0.2	1:01:59.79	25	0	16464	tony
TextEdit	0.1	11:10.02	11	10	261	tony
Safari Networking	0.1	30:04.42	11	0	16458	tony
http://trailers.apple.com, ...	0.1	42:56.93	29	0	16457	tony
launchservicesd	0.1	1:54.25	5	0	78	root
distnoted	0.1	1:02.06	4	0	33284	_spotlight

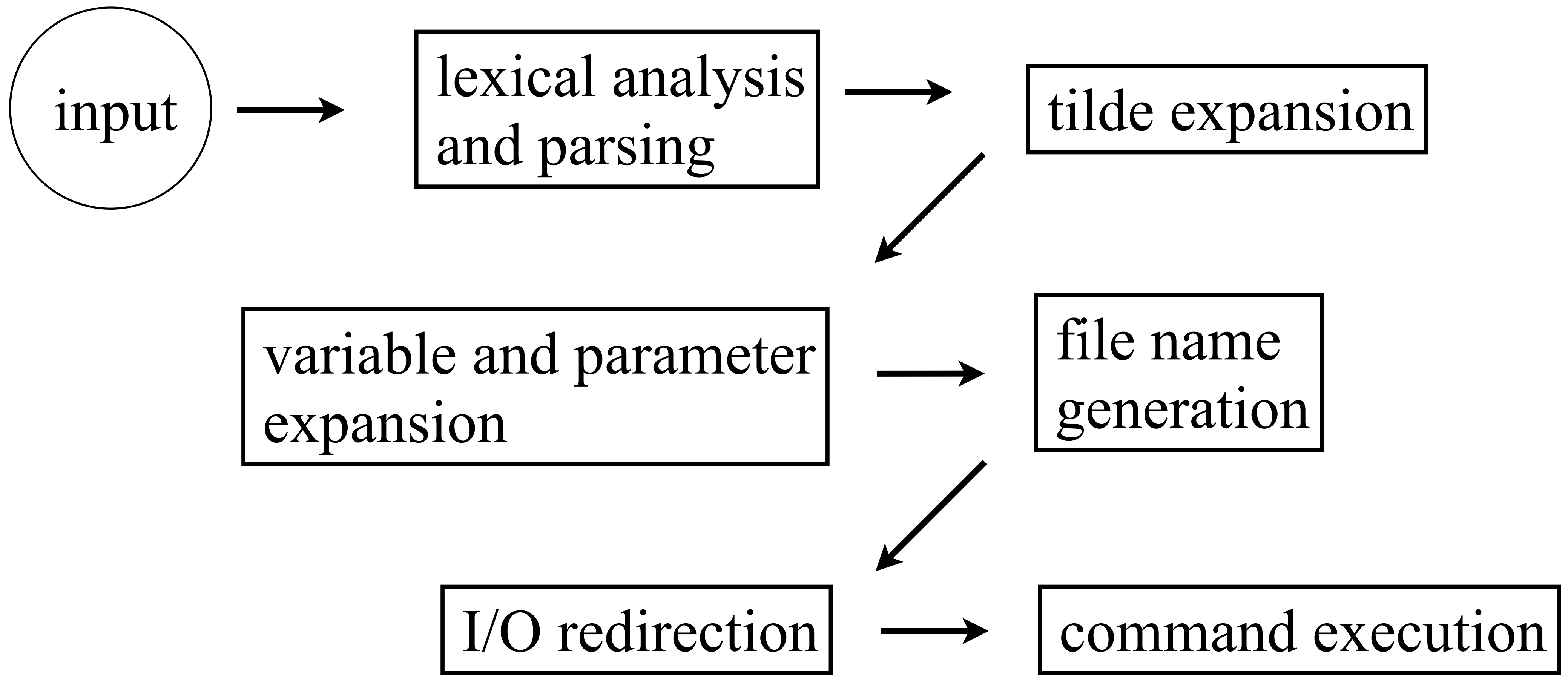
System:	2.81%
User:	2.43%
Idle:	94.75%

Threads:	1717
Processes:	324

Output from Activity Monitor on Mac OS X

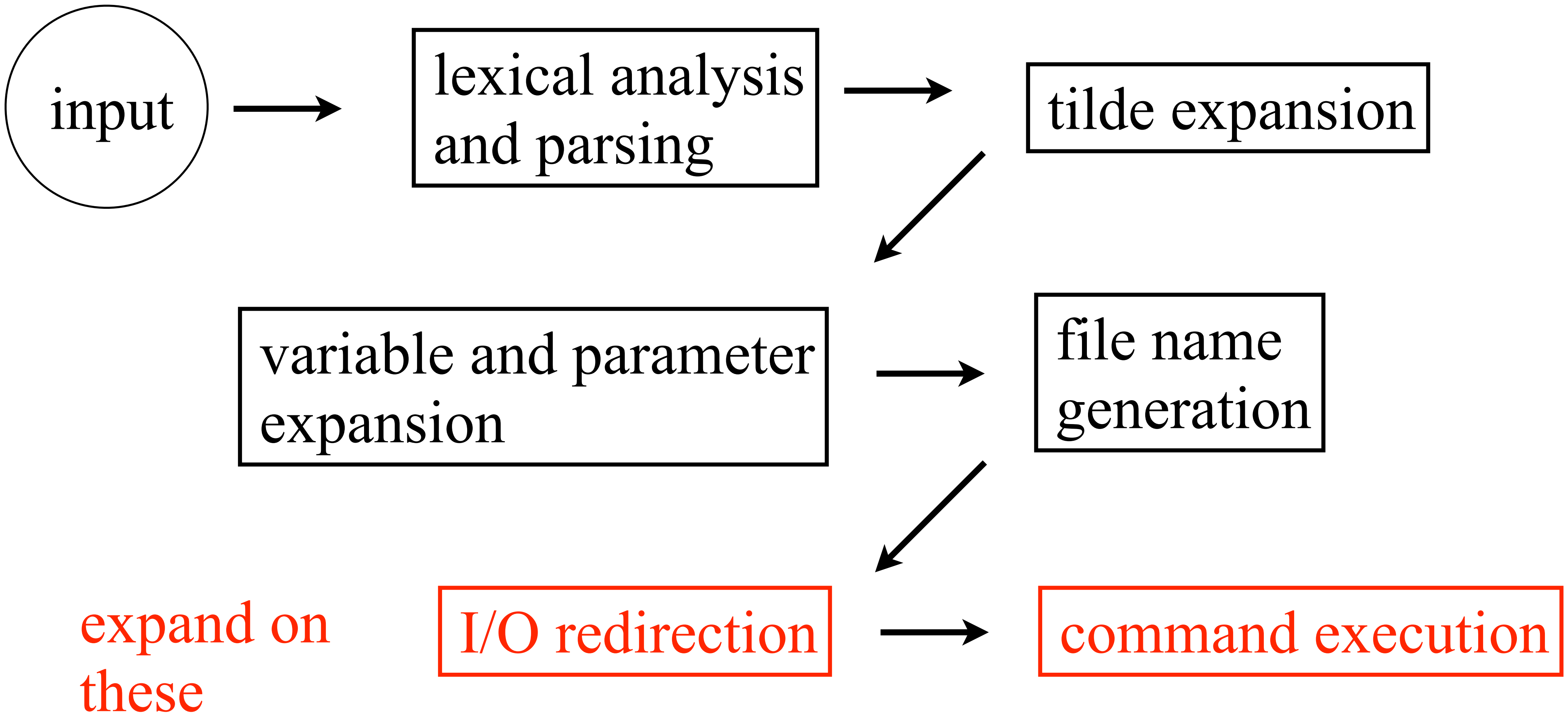
Basic Process Abstraction in UNIX

- process abstraction involved in executing a command from the shell
  - for simplicity many stages not shown



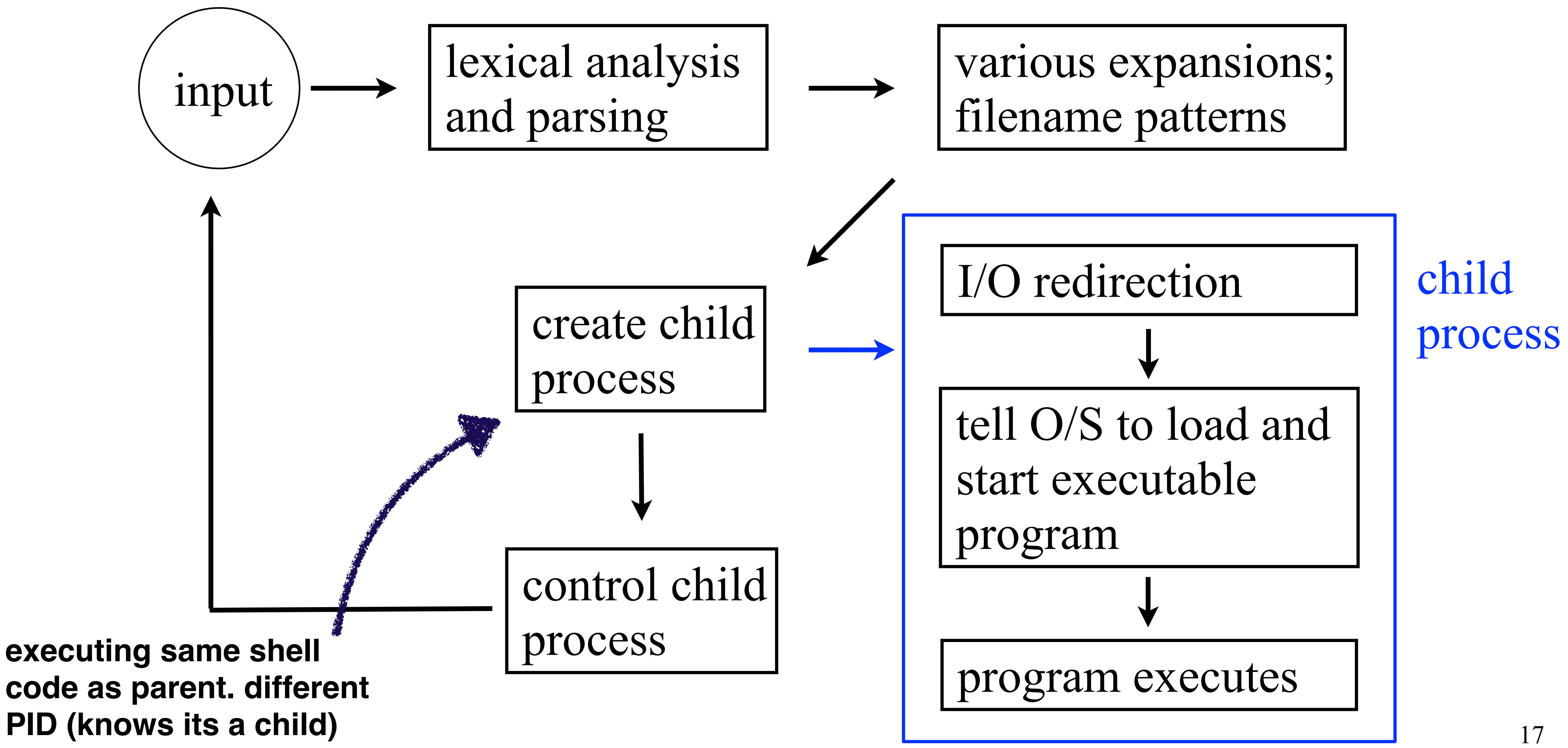
Basic Process Abstraction in UNIX

- process abstraction involved in executing a command from the shell
  - for simplicity many stages not shown



Basic Process Abstraction in UNIX

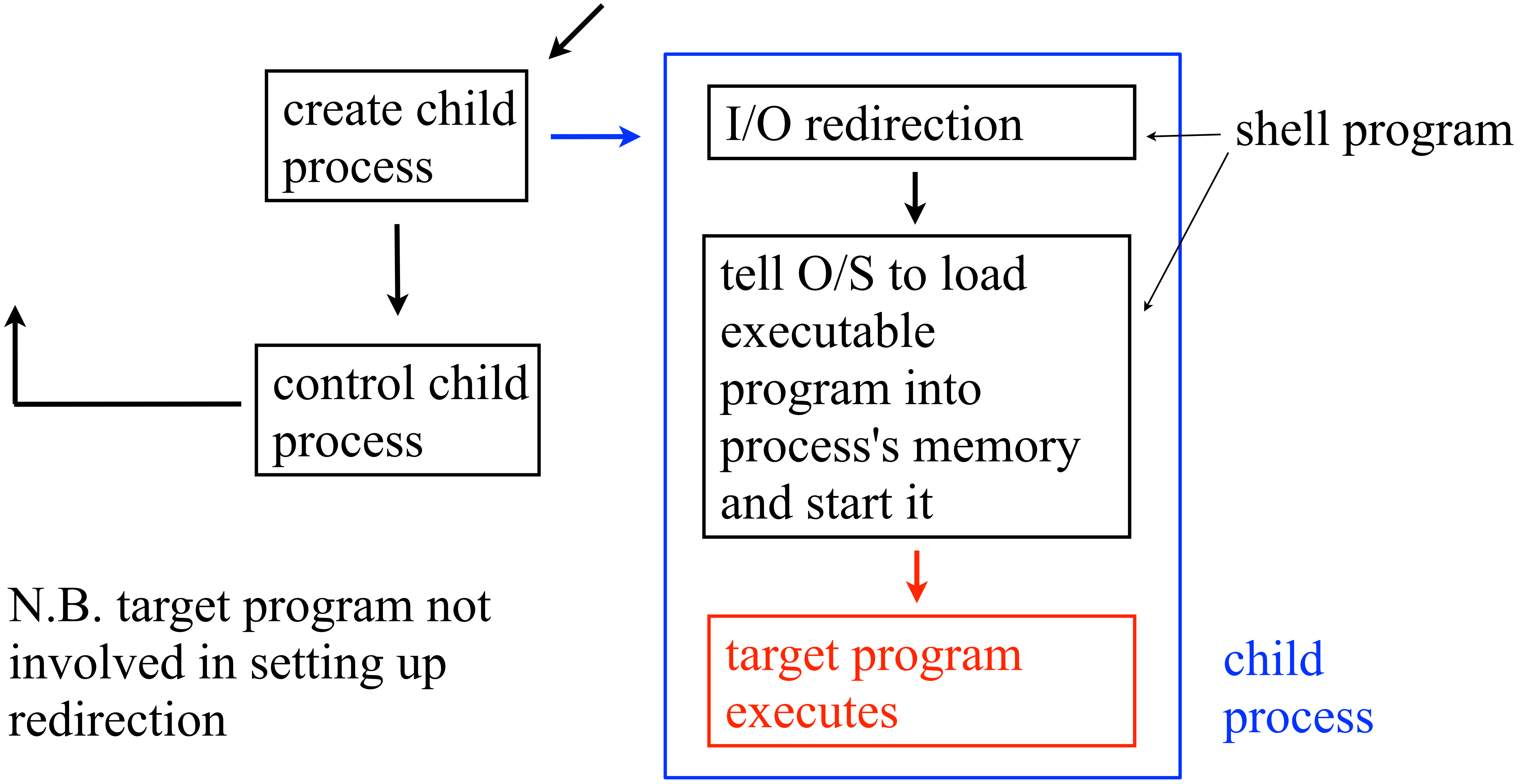
- process abstraction involved in executing a command from the shell



input
lexical analysis and parsing
various expansions; filename patterns
create child process
control child process
I/O redirection
tell O/S to load and start executable program
program executes
child process
executing same shell code as parent. different PID (knows its a child)

Basic Process Abstraction in UNIX

- process abstraction involved in executing a command from the shell



N.B. target program not involved in setting up redirection

Commands Related to UNIX Processes

- list processes
  - `ps`
  - `pstree -h` on `tuxworld` **tells you about all processes running on system**
  - `top`
- `uptime` **how long the system has been up, how many users, load average**
- `w` and `who` **where the users are coming in from**
- `exit` (built-in) and `^D` (end-of-file)

cntr-D

Commands Related to UNIX Processes

- eliminate processes
  - `kill`
  - `man 7 signal`
  - signals generated by keyboard action: `SIGINT`, `SIGQUIT`
  - useful signals for users: `SIGKILL`, `SIGTERM`
  - `/usr/bin/kill` or `/bin/kill` for `bash`

    `kill` built-in for `csh`,
  - `man 1 kill` or `info kill`

Processes and Jobs

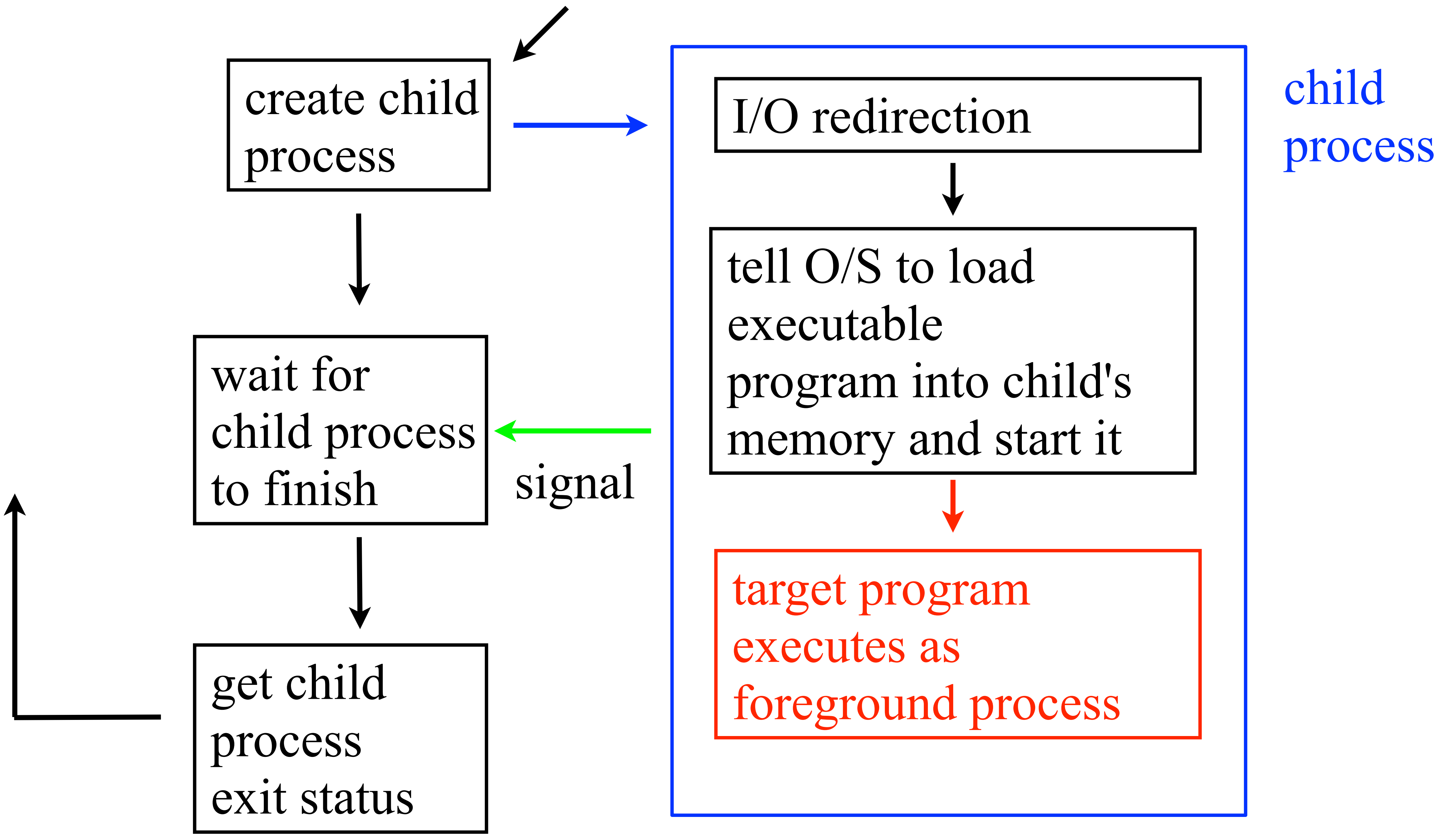
- warning: UNIX shell specific definitions
- *foreground* process:
  - a process that is associated with user input
  - usually means "has control of the keyboard"
  - shell waits for its completion
- *background* process:
  - a process that executes whenever permitted by the OS
  - usually means "does not require user interaction"
  - shell does not wait for its completion

Processes and Jobs

- suspended process:
  - a process that was executing, as permitted by the OS, but is now inactive
  - usually means “was consuming computing resources, but is no longer doing so”. However, the process is still likely using memory resources (e.g. RAM)
  - `ps -l` (LINUX) or `ps -av` (BSD)
- *job*:
  - a suspended or background running process
  - `jobs`

Basic Process Abstraction in UNIX

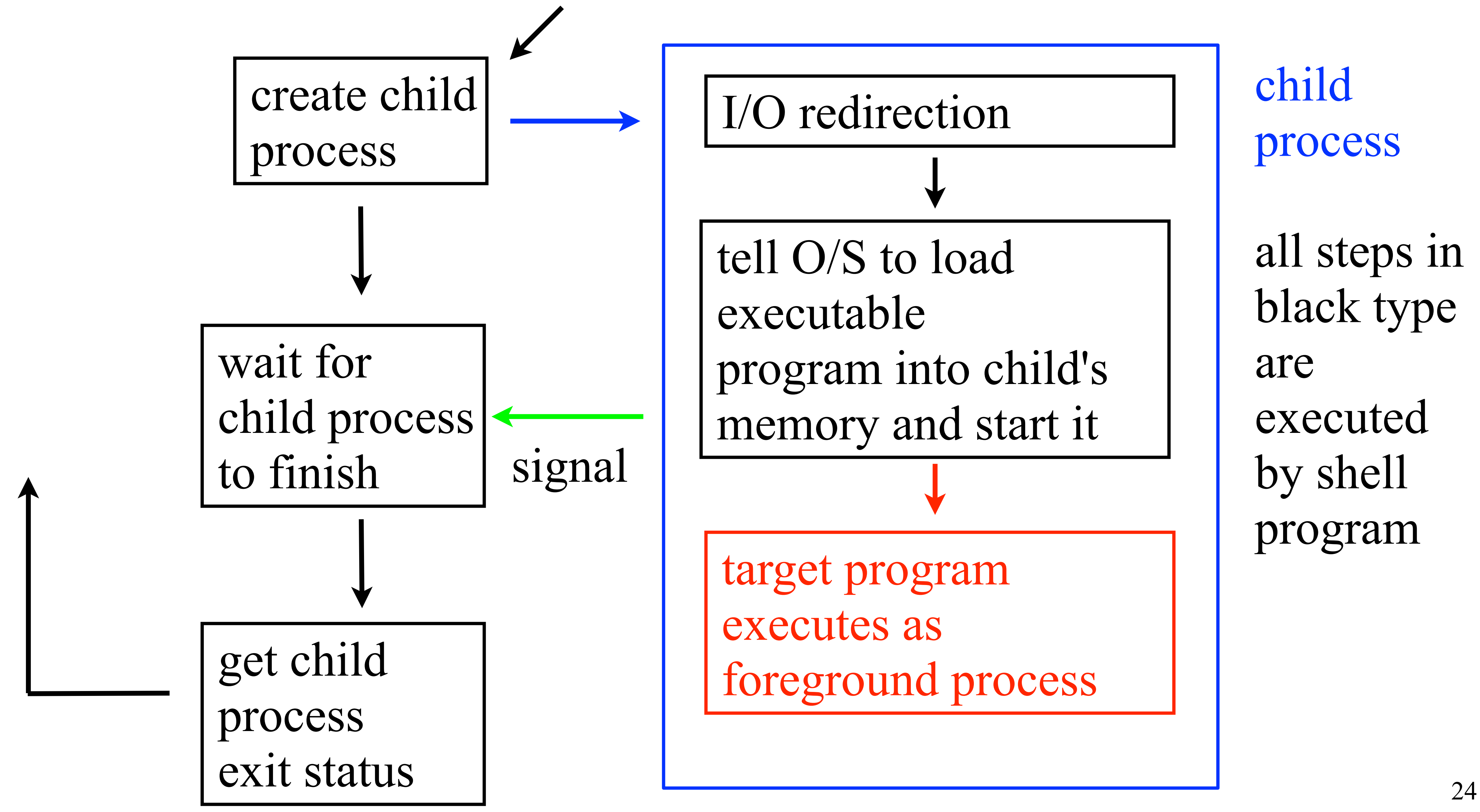
- process abstraction involved in executing a foreground command from the shell



create child process
wait for child process to finish
get child process exit status
signal
I/O redirection
tell O/S to load executable program into child's memory and start it
target program executes as foreground process
child process

Basic Process Abstraction in UNIX

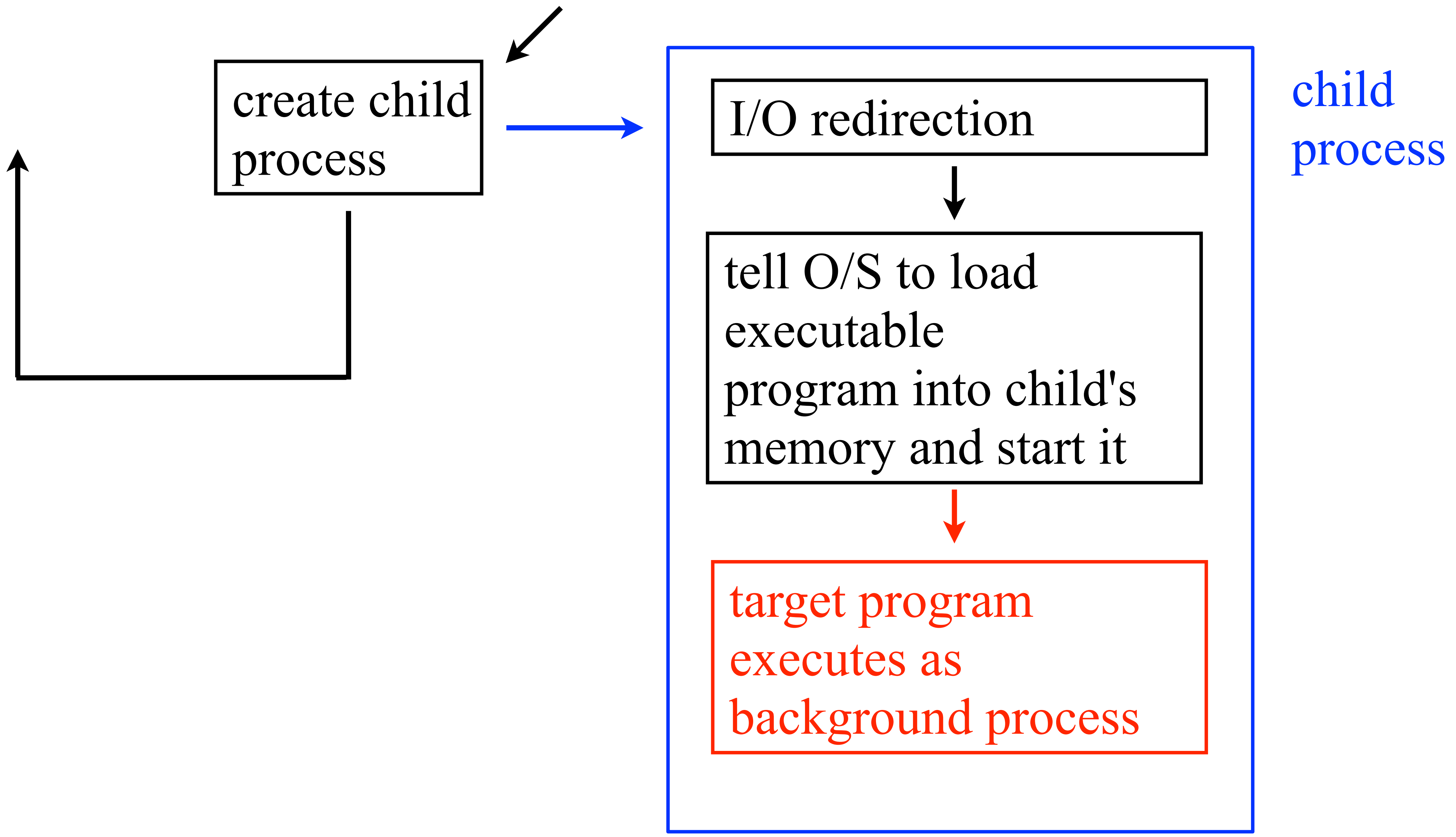
- process abstraction involved in executing a foreground command from the shell



create child process
wait for child process to finish
get child process exit status
signal
I/O redirection
tell O/S to load executable program into child's memory and start it
target program executes as foreground process
child process
all steps in black type are executed by shell program

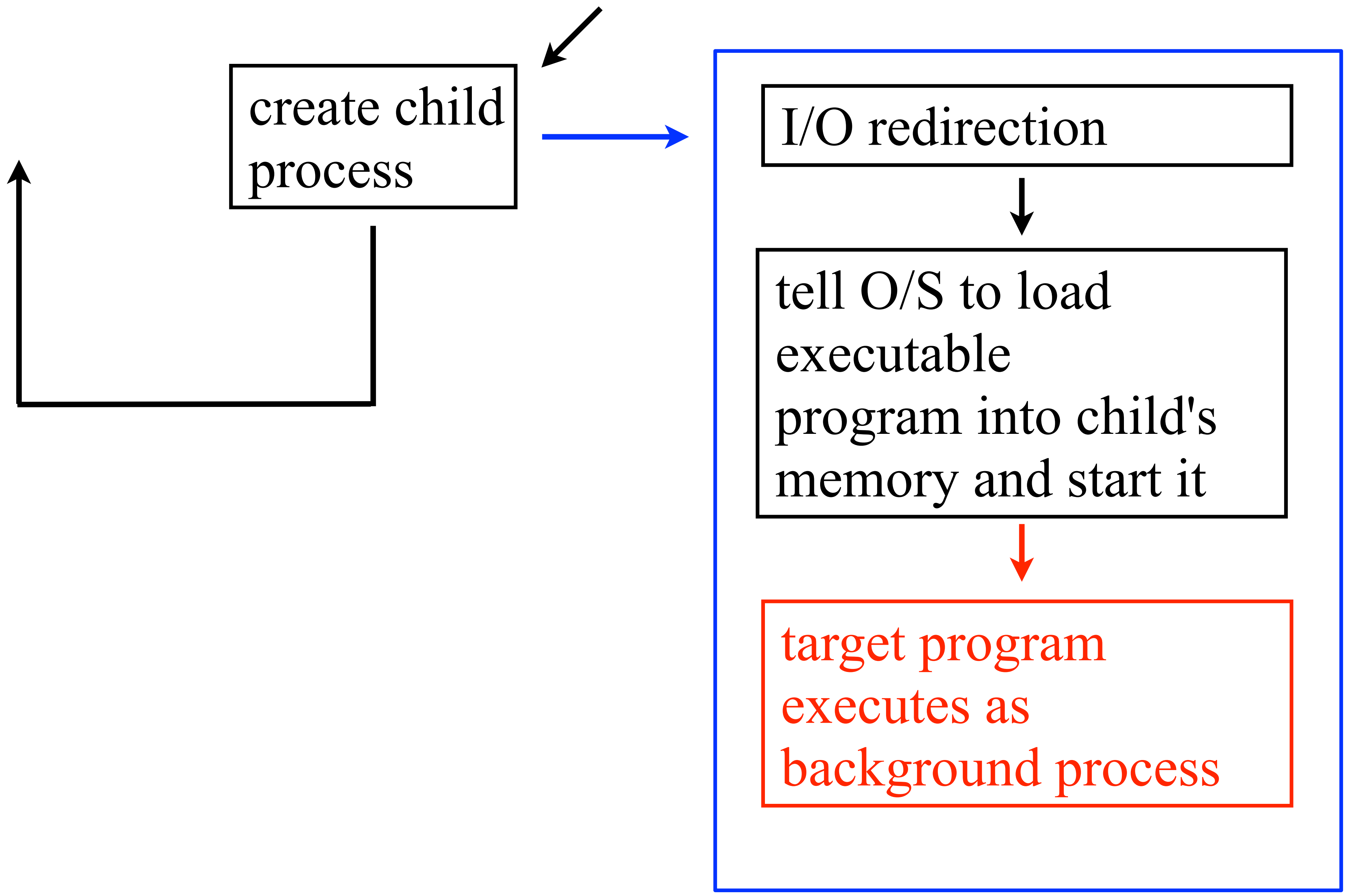
Basic Process Abstraction in UNIX

- process abstraction involved in executing a background command from the shell


create child process
I/O redirection
tell O/S to load executable program into child's memory and start it
target program executes as background process
child process

Basic Process Abstraction in UNIX

- process abstraction involved in executing a background command from the shell

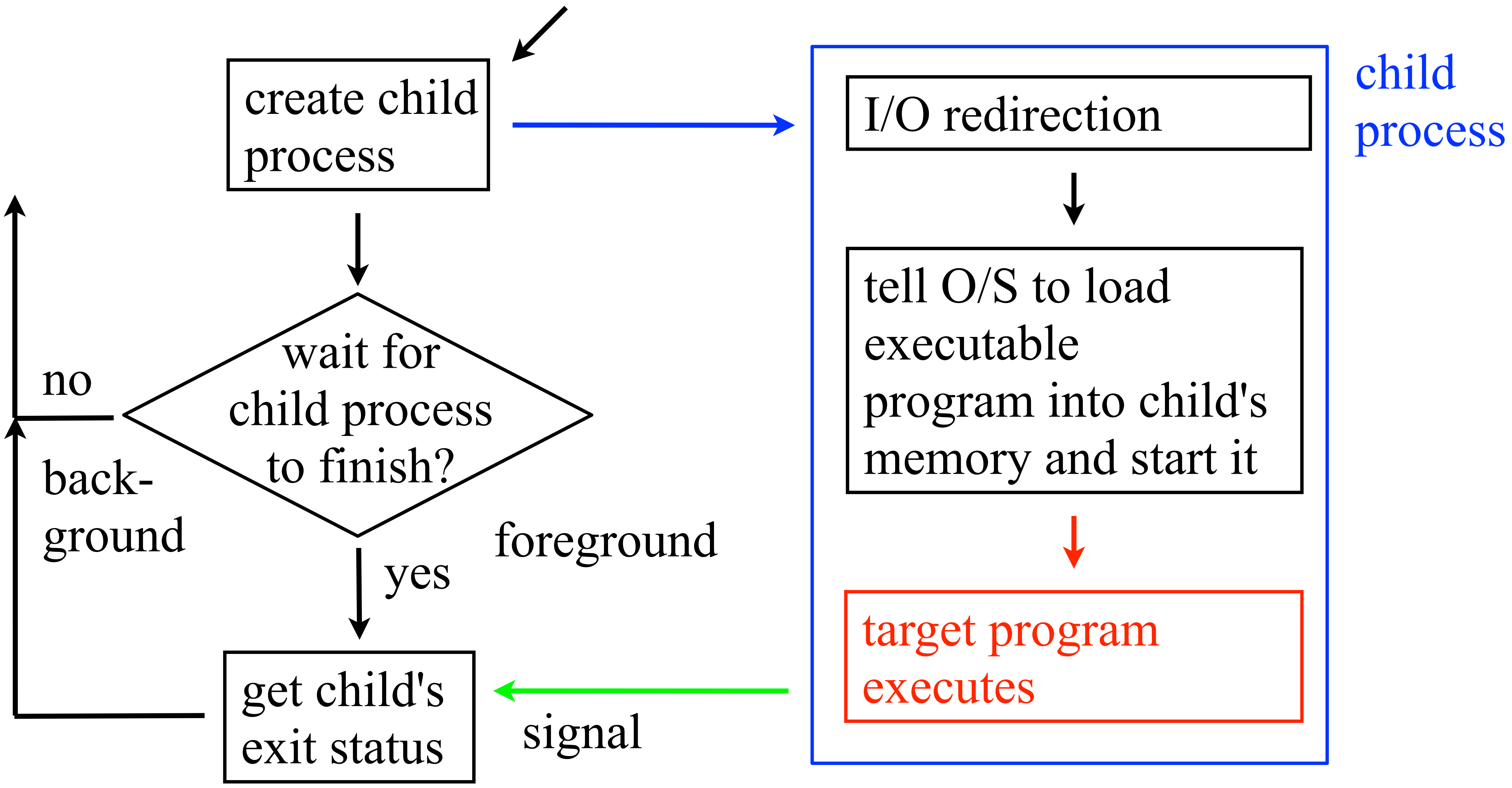

create child process
I/O redirection
tell O/S to load executable program into child's memory and start it
target program executes as background process

child process

all steps in black type are executed by shell program

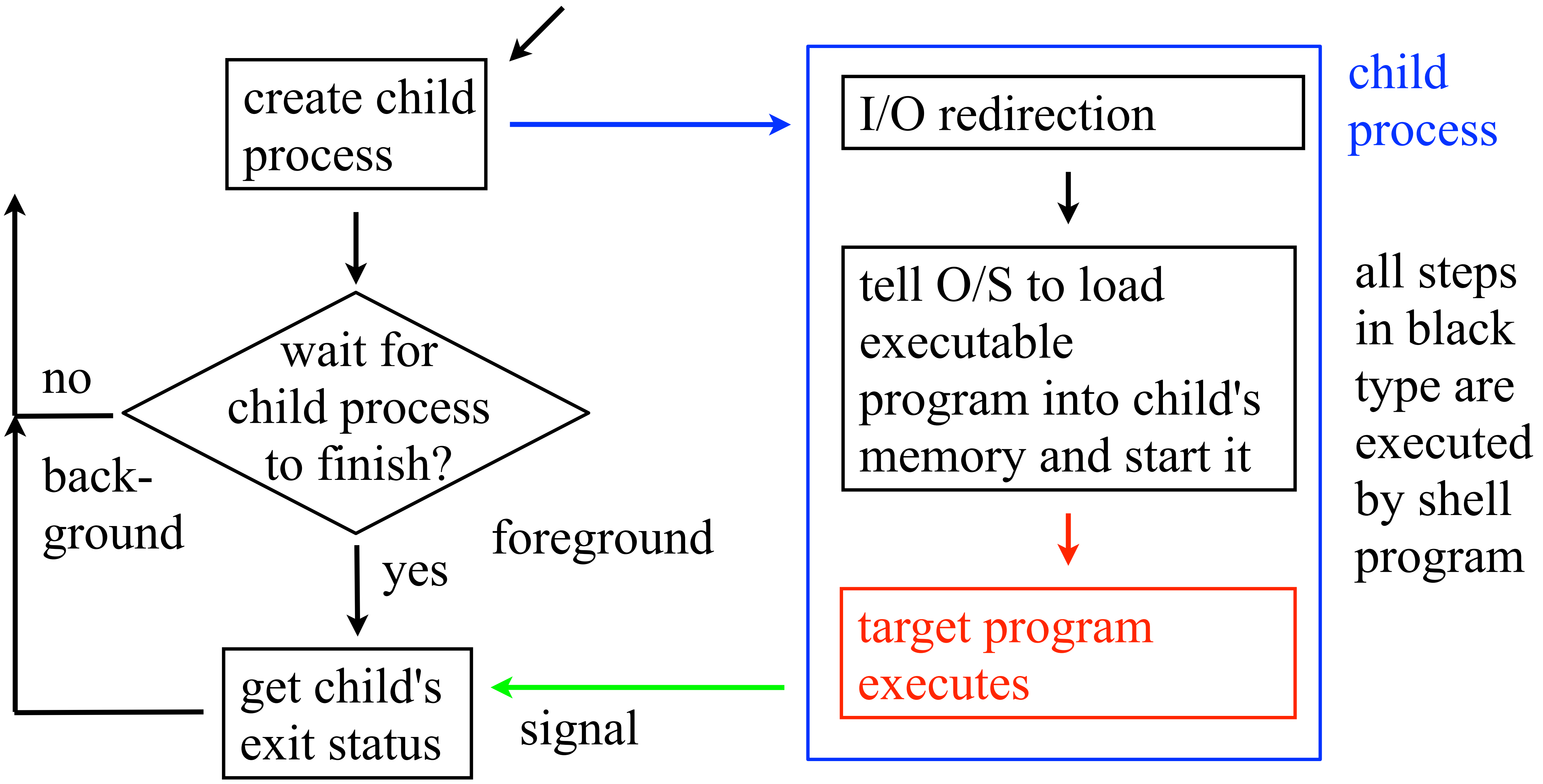
Basic Process Abstraction in UNIX

- process abstraction involved in executing a command from the shell Text



Basic Process Abstraction in UNIX

- process abstraction involved in executing a command from the shell



create child process
wait for child process to finish?
no
back-ground
yes
foreground
get child's exit status
signal
I/O redirection
tell O/S to load executable program into child's memory and start it
target program executes
child process
all steps in black type are executed by shell program

Processes and Jobs

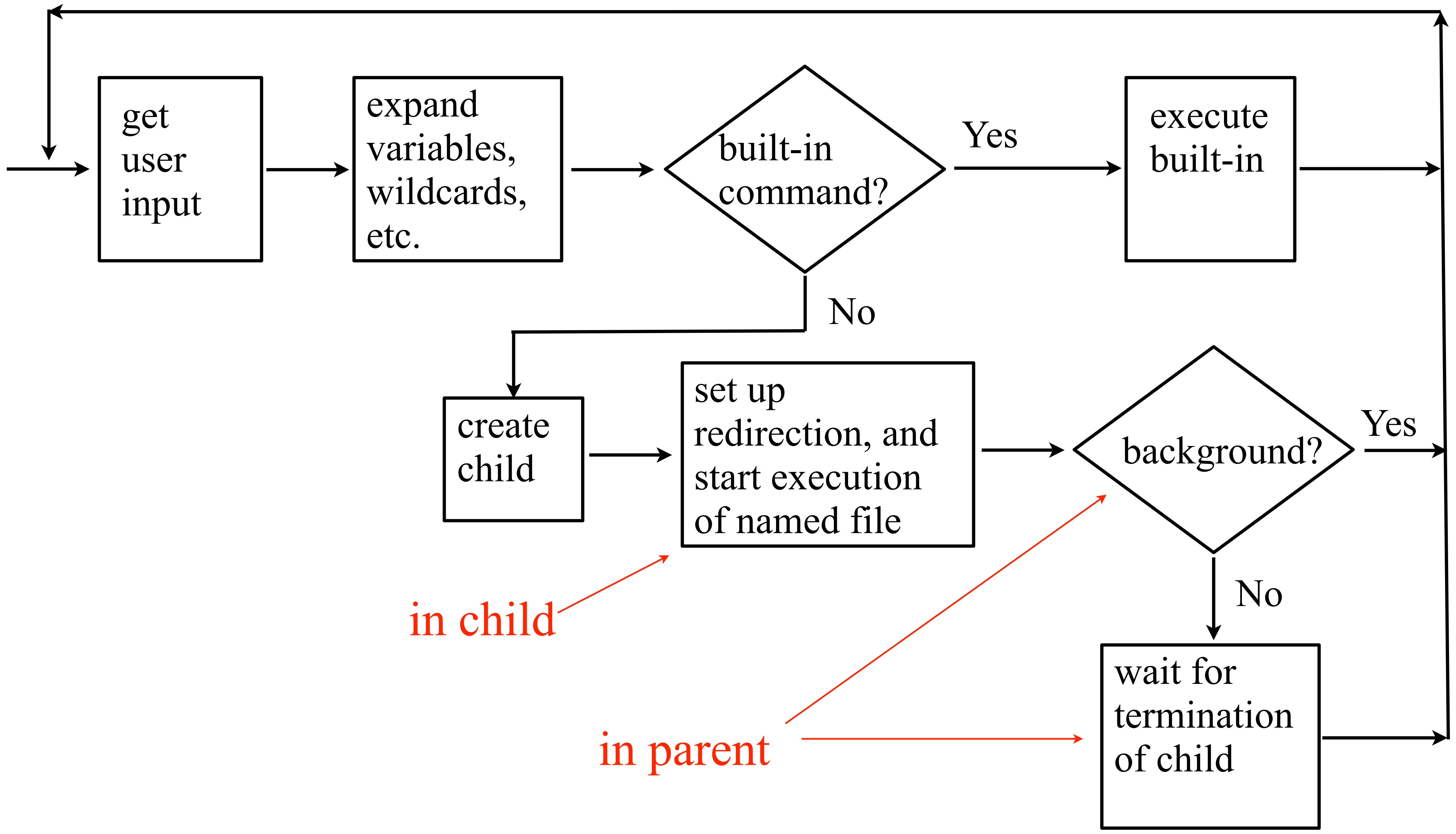
- background process
  - `&`
  - `bg`
- suspend process:
  - `^Z`
- job:
  - `jobs`
  - `%`*n*
- foreground process:
  - `fg %`*n*

Processes and Jobs

- example involving `&`, `kill`, `uniq`

```
# /bin/bash
yes > raw_stuff &
kill -TERM %1
uniq -c raw_stuff
wc -l raw_stuff
rm raw_stuff
```

Basic Shell Operation


get user input
expand variables, wildcards, etc.
built-in command?
Yes
execute built-in
No
create child
set up redirection, and start execution of named file
background?
Yes
No
wait for termination of child
in child
in parent

leave LINUX/UNIX shell … for now

Programming Practice

- programming style
- testing
- software development
  - test-driven design
  - multiple-file development
  - makefiles
  - version control
- debugging
- profiling
- portability

Programming Style

- outline
  - motivation
  - style issues
    - names: variables, constants, macros
    - expressions and statements
    - indentation and spacing
    - idioms
      - loop idioms
    - magic numbers
    - comments
  - defensive programming $\longleftarrow$ later
- reference: *The Elements of C++ Style*

Motivation

- within a project
  - software organizations often impose a particular programming style
  - multiple developers
  - consistent style to maximize readability
- maintenance
  - visual appearance of code affects understandability
- important for understanding your own code

Characteristics of Good Style

- clear and simple
  - straightforward logic
  - conventional (programming) language use
  - natural expression
  - meaningful names
  - neat formatting
  - helpful comments
- consistent

Choice of a Good Style

- many possibilities
- details of choice less important than the fact that you have a style and stick to it
- if working on a program you didn't write, preserve the style you find there

Names

- general principles
  - a name should be informative, concise, memorable, and — if possible — pronounceable

    e.g. `fllcx(SMRHSHSCRTCH, MAXRODDHSH)`
  - the broader the scope of a variable, the more information should be conveyed by its name
    - "use descriptive names for globals, short names for locals"

      e.g. within a tight loop, `n` or `i` are great. However they are not acceptable as global variable names
    - locals used in conventional ways can have very short names

Example

- too much

```
accumulating_count = 0;
for ( theElementIndex = 0; theElementIndex < numberOfElements;
      theElementIndex++ )
  accumulating_count += elementArray[theElementIndex];
```

- too little

```
c = 0;
for ( i = 0; i < n; c += a[i++] );
```

Names

- general principles
  - be consistent
  - give related things related names that show their relationship but highlight their difference
  - be accurate
    - otherwise, can lead to mystifying bugs
    - e.g.

```
#define isoctal(c) ((c) >= '0' && (c) <= '8')
```

Macros are described on pages 240-241 of Schildt text

Names

- general principles
  - use active names for functions

    e.g. `getLine()` rather than `lineReader()`

    - however, name functions that return booleans so that the return value is unambiguous

      e.g. `isOctal()` rather than `checkOctal()`

Names

- examples
  - using names for pointers that begin with or end in 'p'
  - initial capital letters for globals
  - all capital letters for constants
  - when combining words into names
    - concatenate + capitalize, or
    - use '_'
- most important thing: be consistent!

Example

- Exercise: comment on the choice of names and values in the following code

```
#define TRUE 0
#define FALSE 1

if ( (ch = getchar()) == EOF )
  not_eof = FALSE;
```

- better?

"`man 3 getchar`" to find out what `getchar()` does

Example

- Exercise: improve this function

```
bool concat( char *s, char *t ) {
    if(  strcmp( s, t ) < 0 )
        return true;
    else
        return false;
}
```

- better?

"`man 3 strcmp`" to find out what `strcmp()` does

Expressions and Statements

- principles
  - write expressions and statements in a way that makes their meaning as apparent as possible
  - write the clearest code that does the job
    - counterexample in file `2016.09.20.1.cc`

"`?:`" construct described on pages 47-48 of Schildt text

Expressions and Statements

- principles
  - write expressions and statements in a way that makes their meaning as apparent as possible
  - write the clearest code that does the job
    - counterexample in file `2016.09.20.1.cc`
  - use natural form for expressions
    - e.g

```
if( !(block_id < actbliks) ||
    !(block_id >= unblocks) ) …
```

    - should be?

Expressions and Statements

- principles
  - parenthesize to resolve ambiguity
    - sometimes necessary because of operator precedence
    - e.g. `if( x & MASK == BITS )`
    - interpreted as?

Bitwise-AND is described on pages 42-44 of Schildt text

Expressions and Statements

- principles
  - parenthesize to resolve ambiguity
    - sometimes necessary because of operator precedence
    - e.g. `if( x & MASK == BITS )`
    - better?

Expressions and Statements

- principles
  - parenthesize to resolve ambiguity
    - sometimes necessary because of operator precedence
    - e.g. `while( c = getchar() != EOF )`
    - interpretation?

"`man 3 getchar`" to find out what `getchar()` does

Expressions and Statements

- principles
  - parenthesize to resolve ambiguity
    - sometimes necessary because of operator precedence
    - e.g. `while( c = getchar() != EOF )`
    - better?

Expressions and Statements

- principles
  - parenthesize to resolve ambiguity
    - even when not strictly necessary
    - e.g.

```
leap_year = y % 4 == 0 &&
            y % 100 != 0 ||
            y % 400 == 0;
```

    - better?

Expressions and Statements

- principles
  - break up complex expressions
    - e.g. `*x += (*xp=(2*k<(n-m)?c[k+1]:d[k--]));`
    - better?

Expressions and Statements

- principles
  - be clear
    - write clear code, not clever code
    - clarity is not the same as brevity

      e.g. `child=(!LC&&!RC)?0:(!LC?RC:LC);`

  - be careful with side-effects
    - the order of execution of side-effects is undefined in C

      e.g. `str[i++] = str[i++] = ' ';`

      e.g. in example `2016.09.20.2.c`

Example

- Exercise: improve this fragment

```
length = (length < BUFSIZE) ? length : BUFSIZE;
```

- better?

Example

- Exercise: how is this code excerpt problematic?

```
int read( int *ip) {
    scanf( "%d", ip );
    return( *ip );
}
...
insert( &graph[vert], read( &val ), read( &ch ) );
```

Indentation and Spacing

- principles
  - use spaces around operators to suggest grouping
  - indent to show structure
    - e.g.

```
for(n=0;n<100;field[n++]='\0');
*i='\0'; return( '\n' );
```

    - better?

Indentation and Spacing

- principles
  - use spaces around operators to suggest grouping
  - indent to show structure
  - however, sprawling layouts detract from readability
  - use a consistent indentation and brace style
    - syntax-driven editing tools help

Example

- mixture of tabs and spaces
- are the names well-chosen?

```
          // binary search function in C++
          int bs(int v, const int *a, const int n) {
     tab→      if (n<=0)
4 spaces→     return -1;
            int m=n/2;
               if (a[m]>v)
              return bs(v,a,m-1);
            else if (a[m]<v)
                     return bs(v,a+m+1,n-m-1);
            else
              return m;
          }
```

Example

- better?