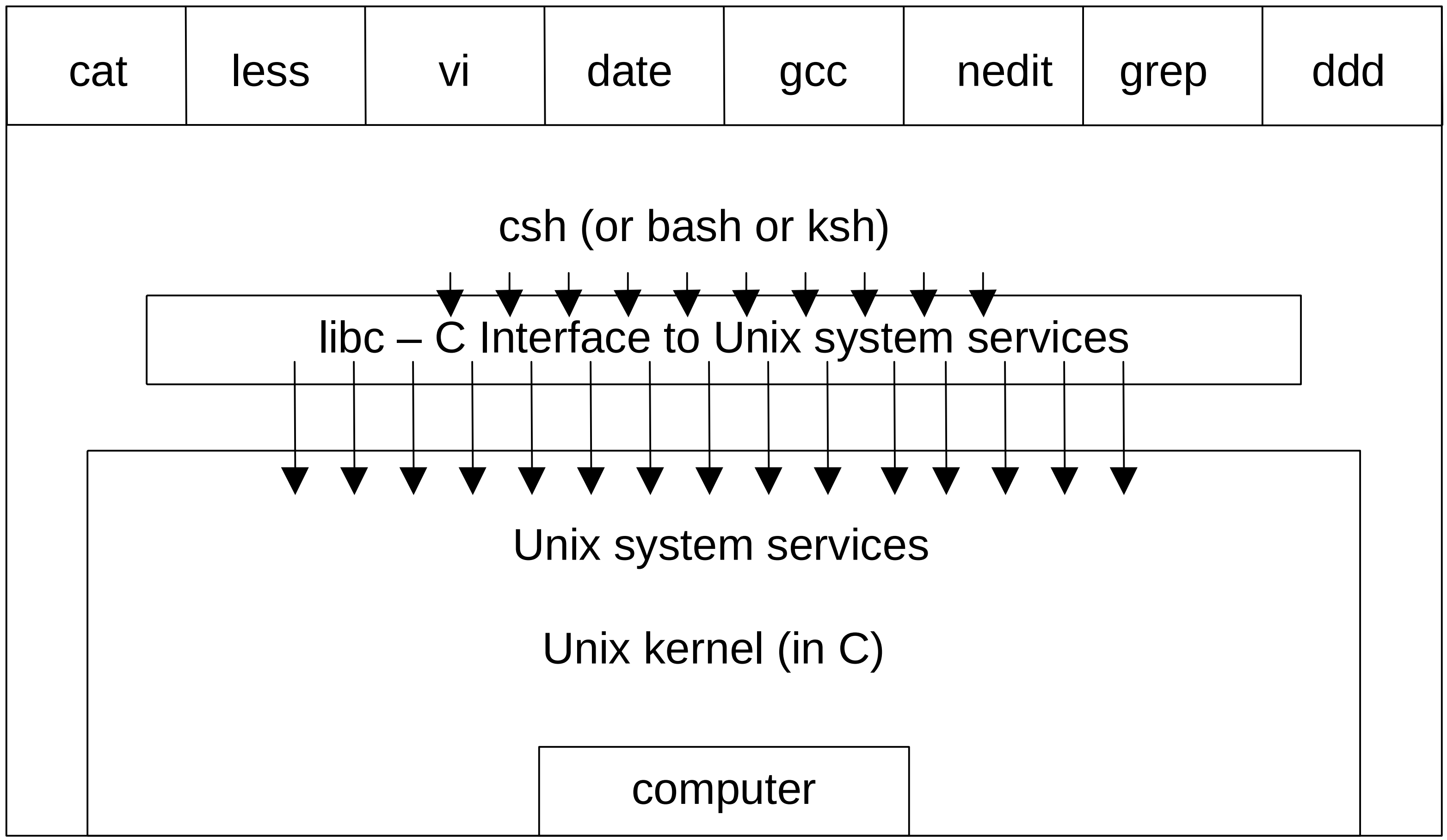
Compilers, Interpreters, Libraries

Comparing compilers and interpreters

Shared vs. non-shared libraries.

Layers of System Software



Compiler vs. Interpreter

- Somehow we need to convert a program into machine code (object code).
- A compiler passes over a whole program before translating it into object code.
- An interpreter reads and executes one line of code at a time.
- An interpreter is a compiled program (often written in C).

C/C++ compiler

- *Preprocessor* does text replacement
  - `#include` replaced by the text of the included file.
  - `#define` macros replaced throughout each file.
- *Compiler* parses the program, performs optimization, and produces assembly code.
- *Assembler* translates assembly code into machine code.
- *Linker* combines object files and libraries into an executable file. It resolves any remaining symbol references.

Java Compiler/Interpreter

- *Compiler* translates program to byte code.
- The *JVM* is a byte code interpreter that translates byte code to machine code.
- Byte codes implement fine grain primitives. They are generic enough that other languages may be compiled to Java byte code.

Shell Interpreter

- The interpreter is a C program!
- The shell interpreter is the program executed when you write

```
#!/bin/sh
```

- Each line of a shell script is input to a C program that parses the line, and determines how to execute it.

Standard Libraries

- System calls are not part of the C language definition.
- System calls are defined in libraries (.a .so)
- Libraries typically contain many .o object files.
- To create your own library archive file:

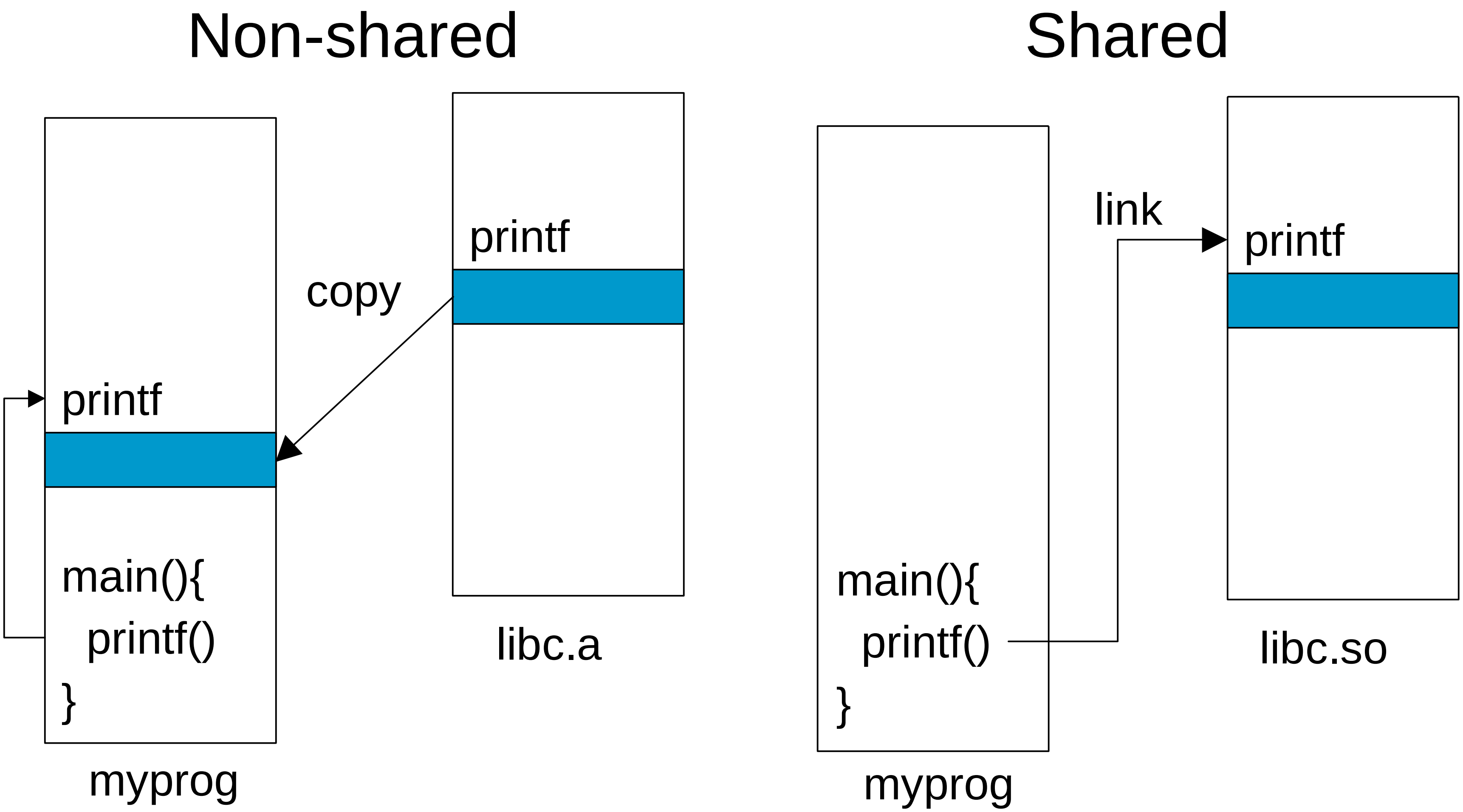
```
ar crv mylib.a *.o
```

- Look in `/usr/lib` and `/usr/local/lib` for system libraries.

Shared Libraries

- .a libraries are not shared. The functions used are copied into the executable of your program.
  - size bloat when lots of processes use the same libraries
  - performance and portability are the wins
- .so libraries are shared. One copy exists in memory, and all programs using that library link to it to access library functions.
  - reduces total memory usage when multiple processes use the shared library.
  - small performance hit as extra work must be done either when a library function is called, or at the beginning.
  - many tradeoffs and variations between OS's

Shared vs. Non-Shared Libraries



System calls

- Perform a subroutine call directly to the Unix kernel.
- `libc` provides the C interface to system calls
- 4 main categories
  - File management
  - Process management
  - Communication
  - Error handling

Executing a Program

- A special start-up routine (`crt0`) is always linked in with your program.
- This routine reads the arguments and calls main.
- The `libc` library is automatically linked into your program, which is how you have access to many C functions (`printf`, `open`, etc.)
- Your program also calls special functions on exit that close file descriptors and clean up other resources.

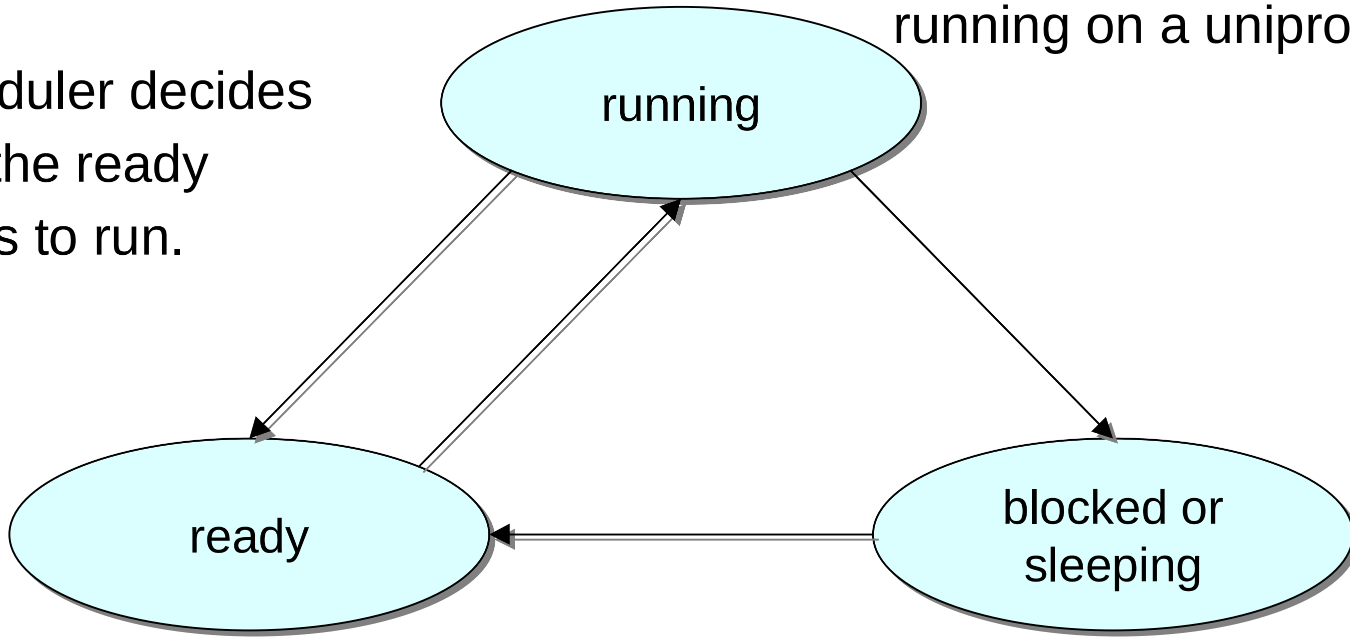
Processes

Creating and using multiple processes

Process State

Only one process can be running on a uniprocessor

The scheduler decides which of the ready processes to run.

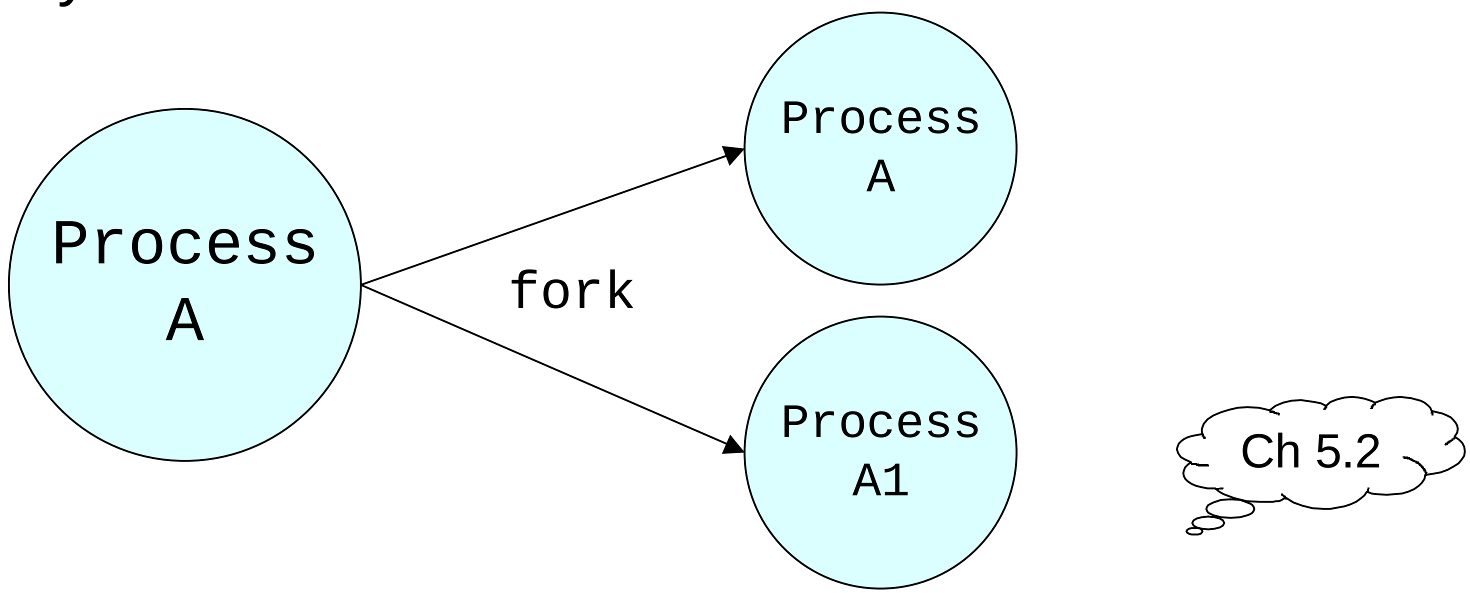


A process is ready if it could use the CPU immediately.

A process is blocked if it waiting for an event (I/O, signal)

Fork

- The fork system call creates a duplicate of the currently running program.
- The duplicate (child process) and the original (parent process) both proceed from the point of the fork with exactly the same data.
- The only difference is the return value from the fork call.



Fork example

```
int main ()
{
   pid_t pid;
   pid = fork();
   if (pid < 0) {
      perror("fork()");
   } else if (pid > 0) {
      printf("parent\n");
   } else {  /* pid == 0 */
      printf("child\n");
   }
   return 0;
}
```

Fork: PIDs and PPIDs

- System call: `int fork(void)`
  - If `fork()` succeeds it returns the child PID to the parent and returns 0 to the child;
  - If `fork()` fails, it returns -1 to the parent (no child is created) and sets `errno`
- Related system calls:
  - `int getpid()` – returns the PID of current process
  - `int getppid()` – returns the PID of parent process (ppid of 1 is 1)

When `fork()` fails

- There is a limit to the maximum number of processes a user can create.
- Once this limit is reached, subsequent calls to `fork()` return -1.

`fork()` properties

- Properties of parent inherited by child:
  - UID, GID
  - controlling terminal
  - CWD, root directory
  - signal mask, environment, resource limits
  - shared memory segments
- Differences between parent and child
  - PID, PPID, return value from fork()
  - pending alarms cleared for child
  - pending signals are cleared for child

Fork example

```
int i;
pid_t pid;

i = 5;
printf("%d\n", i);
pid = fork();

if (pid > 0)
  i = 6; /* only parent gets here */
else if (pid == 0)
  i = 4; /* only child gets here */
printf("%d\n", i);
```

Fork Example

Original process (parent)

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);
/* prints 5 */
pid = fork();
/* pid == 677 */
if (pid > 0)
    i = 6;
else (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 6 */
```

Child process

```
int i; pid_t pid;
i = 5;
printf("%d\n", i);

pid = fork();
/* pid == 0 */
if (pid > 0)
    i = 6;
else if (pid == 0)
    i = 4;
printf("%d\n", i);
/* prints 4 */
```

PID/PPID Example

```
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main() {
   pid_t pid;
   printf("ORIG: PID=%d PPID=%d\n",
          getpid(), getppid());
   pid = fork();
   if (pid > 0)
       printf("PARENT: PID=%d PPID=%d\n",
              getpid(), getppid());
   else if (pid == 0)
       printf("CHILD: PID=%d PPID=%d\n",
                getpid(), getppid());
   return(0);
}
```

Process Termination

Ch 5.6

- Orphan process:
  - a process whose parent is the init process (PID 1) because its original parent died before it did.
- Terminating a process: `exit()`
- Every normal process is a child of some parent, a terminating process sends its parent a SIGCHLD signal and waits for its termination status to be accepted.
- The Bourne shell stores the termination code of the last command in $?.

`wait()`

Ch 5.7

- System call to wait for a child
  - `int wait(int *status)`
- A process that calls `wait()` can:
  - block (if all of its children are still running)
  - return immediately with the termination status of a child (if a child has terminated and is waiting for its termination status to be fetched)
  - return immediately with an error (if it doesn't have any child processes.)

Zombies

- A zombie process:
  - a process that is "waiting" for its parent to accept its return code
  - a parent accepts a child's return code by executing `wait()`
  - shows up as `Z` in `ps –a`
  - A terminating process may be a (multiple) parent; the kernel ensures all of its children are orphaned and adopted by init.

wait and waitpid

- `wait()` can
  - block
  - return with termination status
  - return with error
- If there is more than one child `wait()` returns on termination of any children
- `waitpid` can be used to wait for a specific child pid.
- `waitpid` also has an option to block or not to block

wait and waitpid

- `waitpid` has an option to block or not to block
- `pid_t waitpid(pid, &status, option);`
  - if `pid == -1` → wait for any child
  - if `option == WNOHANG` → non-blocking
  - if `option == 0` → blocking

- `waitpid(-1, &status, 0);`
  is equivalent to `wait(&status);`

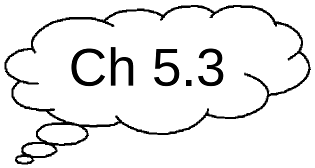
Example of wait

```
#include <sys/types.h>
#include <sys/wait.h>
int main(void) {
  int status;
  if(fork() == 0) exit(7); /*normal*/
  wait(&status); prExit(status);
  if(fork() == 0) abort(); /*SIGABRT*/
  wait(&status); prExit(status);
  if(fork() == 0) status /= 0; /*FPE*/
  wait(&status) prExit(status);
}
```

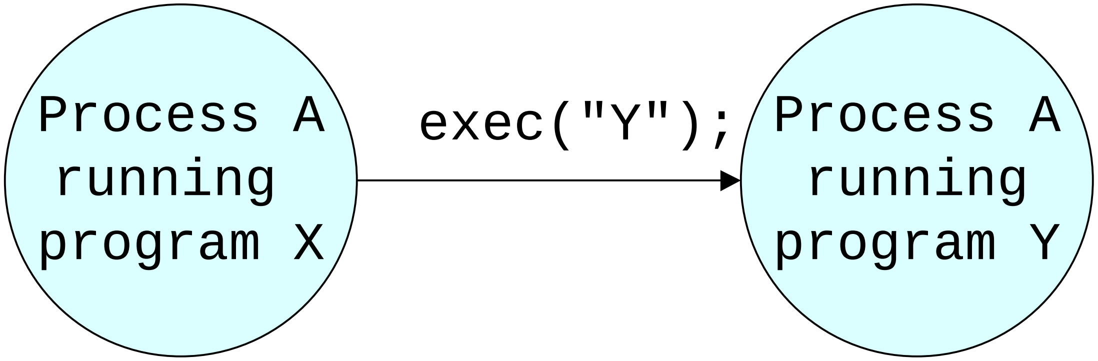
prExit.c

```
#include <sys/types.h>
void prExit(int status) {
  if(WIFEXITED( status ) )
      printf("normal termination\n");
  else if(WIFSTOPPED( status ))
      printf("child stopped, signal no.= %d\n",
            WSTOPSIG(status));
  else if(WIFSIGNALED( status ) )
      printf("abnormal termination, "
         "signal no.= %d\n", WTERMSIG(status));
}
```

Exec



- The exec system call replaces the program being run by a process by a different one.
- The new program starts executing from the beginning.
- On success, exec never returns, on failure, exec returns -1.



Exec example

Program X

```
int i = 5;
printf("%d\n", i);
exec("Y");
printf("%d\n", i);
```

Program Y

```
printf("hello\n");
```

exec properties

- New process inherits from calling process:
  - PID and PPID, real UID, GID
  - controlling terminal
  - CWD, root directory, resource limits
  - pending signals
  - pending alarms

exec()

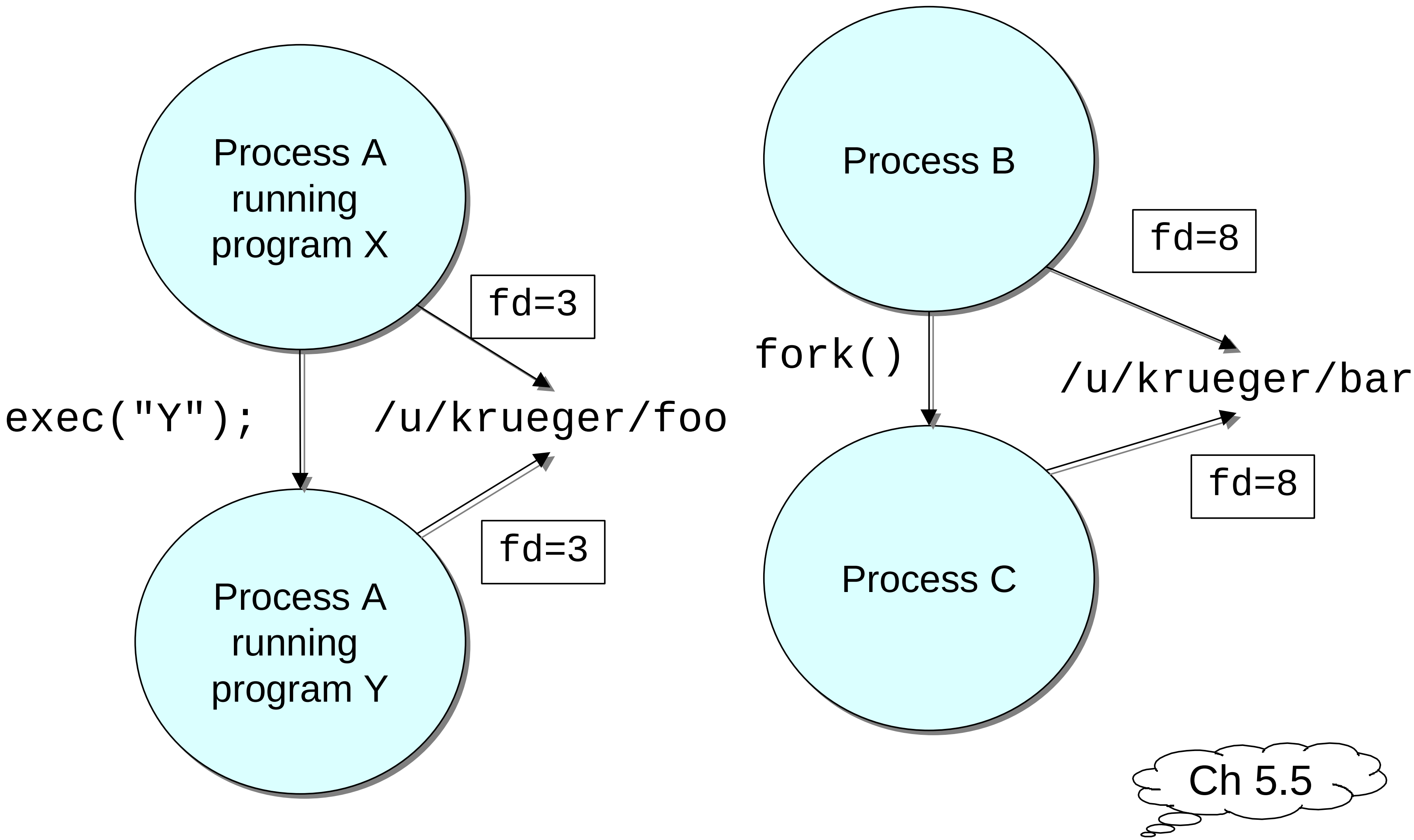
- Six versions exec():

```
execl(char *path, char *arg0, …, (char *)0);
execv(char *path, char *argv[]);
execle(char *path, char *arg0, …,(char *)0,
       char *envp[]);
execve(char *pathname, char *argv[],
       char *envp[]);
execlp(char *file, char *arg0, …,(char *)0);
execvp(char *file, char *argv[]);
```

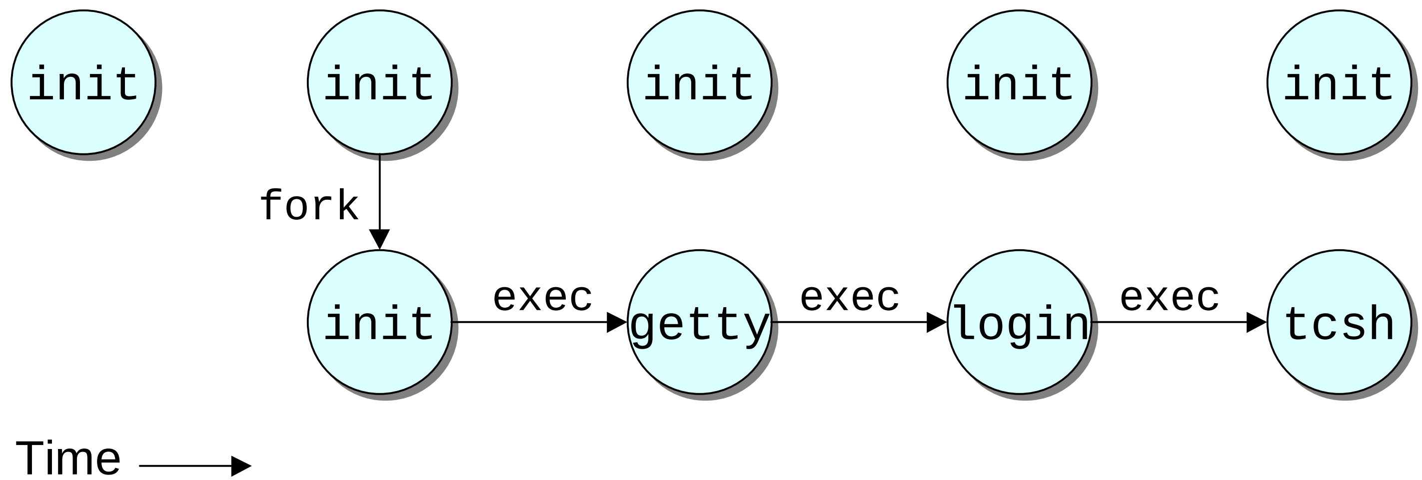
Processes and File Descriptors

- File descriptors are handles to open files.
- They belong to processes not programs.
- They are a process’s link to the outside world.

FDs preserved across fork and exec

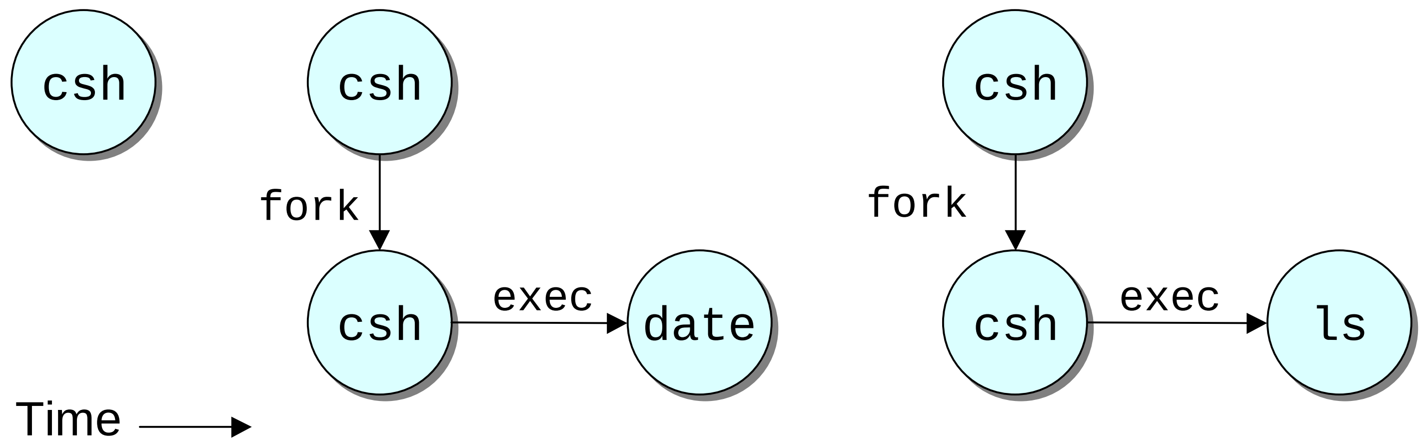


Initializing Unix



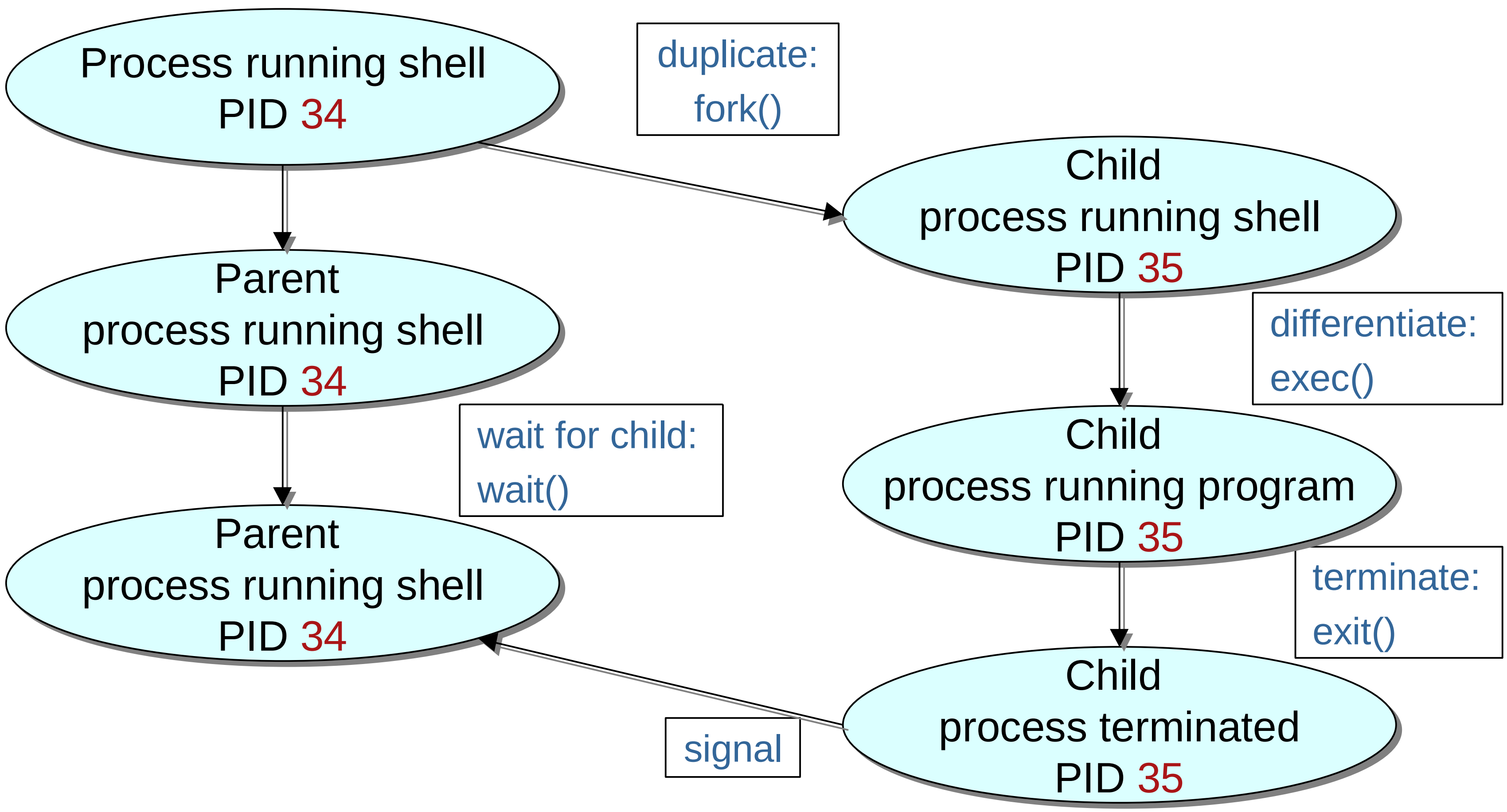
- See “top”, “ps –aux” to see what’s running
- The **only** way to create a new process is to duplicate an existing process. Therefore the ancestor of **all** processes is `init` with pid = 1

How csh runs commands



- When a command is typed, csh forks and then execs the typed command.
- After the fork, file descriptors 0, 1, and 2 still refer to stdin, stdout, and stderr in the new process.
- By convention, the executed program will use these descriptors appropriately.

How csh runs


Process running shell
PID 34
duplicate:
fork()
Child
process running shell
PID 35
Parent
process running shell
PID 34
differentiate:
exec()
wait for child:
wait()
Child
process running program
PID 35
Parent
process running shell
PID 34
terminate:
exit()
Child
process terminated
PID 35
signal