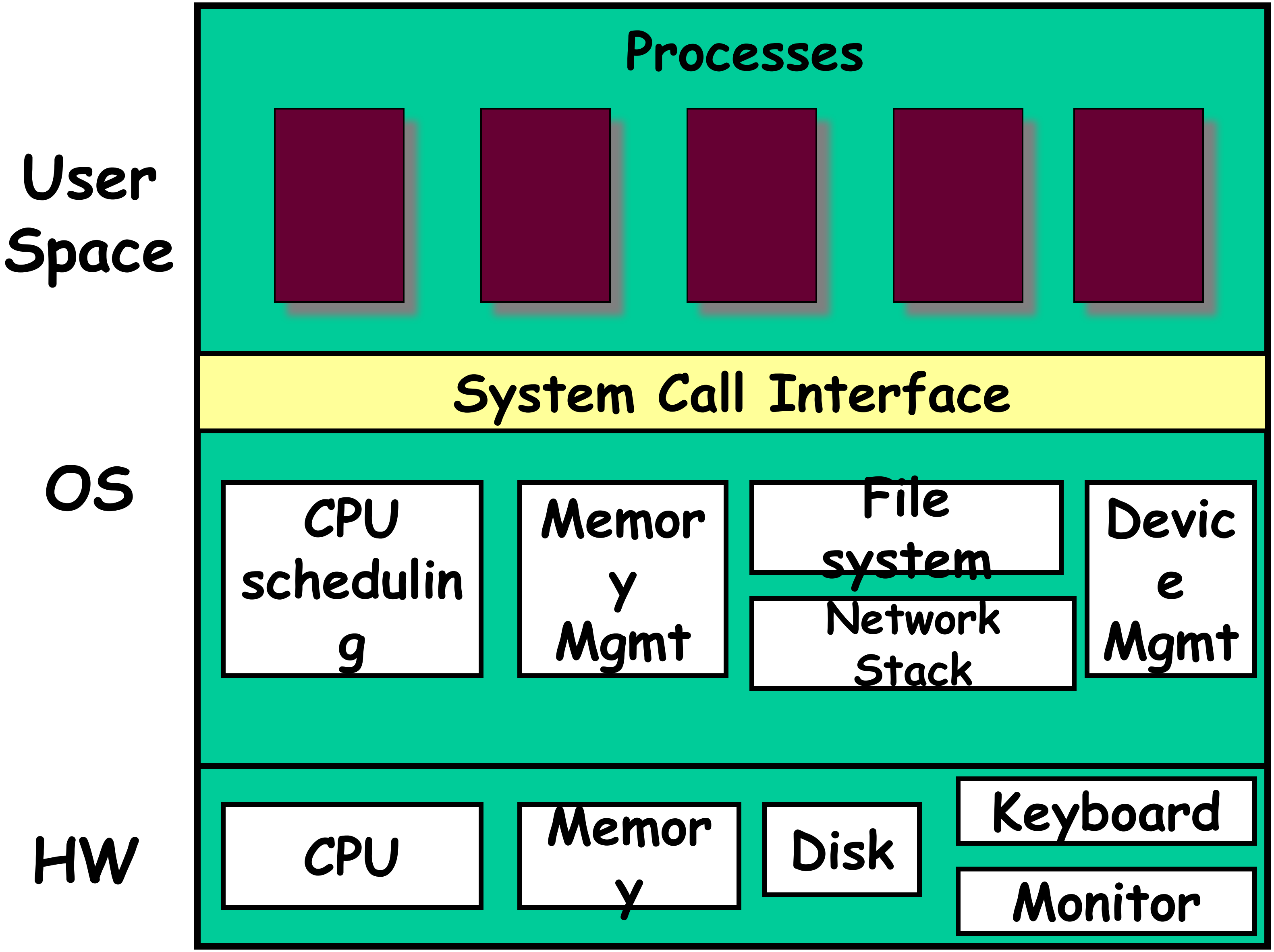
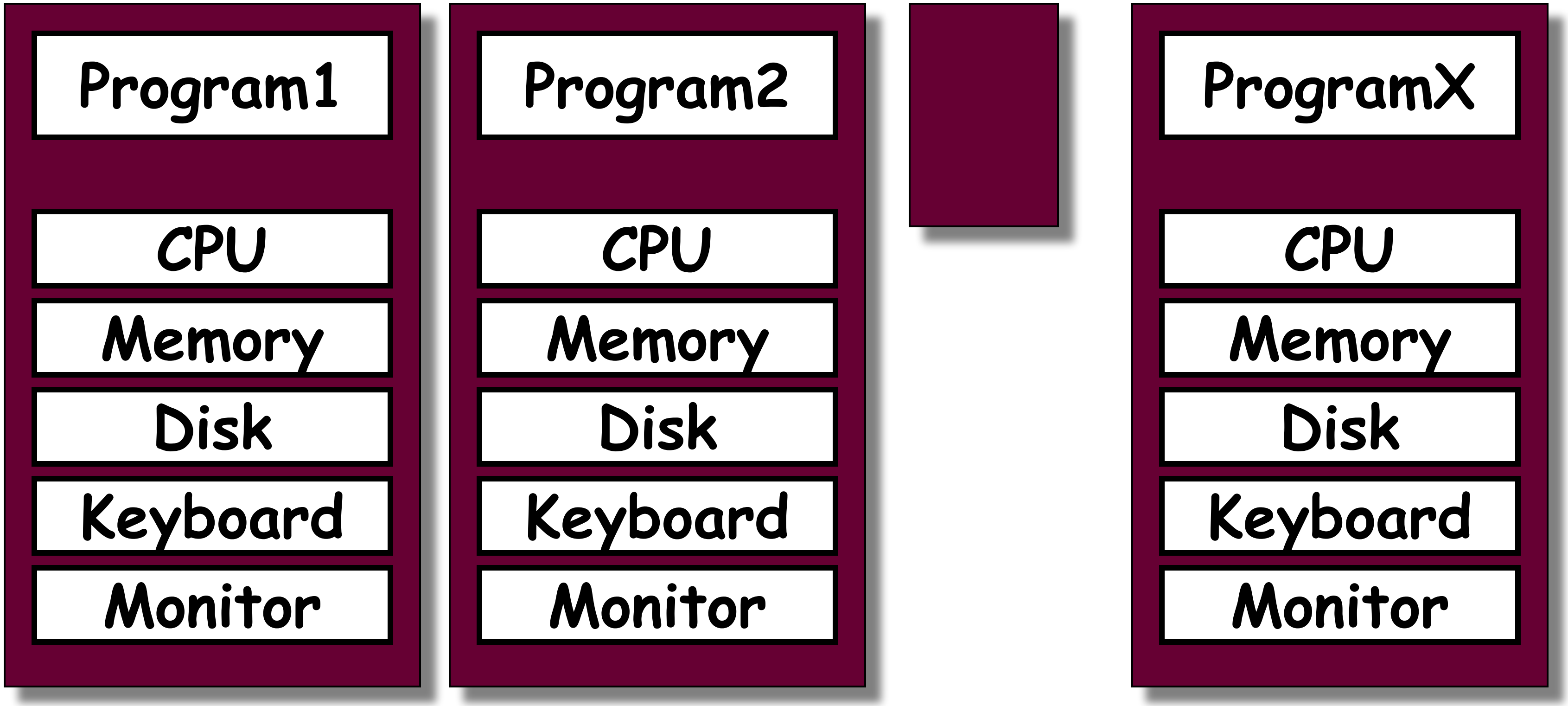
Today’s Topics

- Introducing process: the basic mechanism for concurrent programming
  - Process management related system calls
    - Process creation
    - Process termination
    - Running another program in a process
    - Synchronization between Parent/child processes

Computer systems Overview



Computer systems user's view



Each program owns its own (virtual) computer. The execution of a program does not affect one another.

Process

- Informal definition:

  A process is a program in execution.

- Process is not the same as a program.
  - Program is a passive entity stored in disk
  - Program (code) is just one part of the process.

What else in a process?

- Process context – everything needed to run resume execution of a program:
  - Memory space (static, dynamic)
  - Procedure call stack
  - Open files, connections
  - Registers and counters :
    - Program counter, Stack pointer, General purpose registers
  - ……

Why process?

- Multiple processes (users) share the system resources.
- Multiple processes run independently

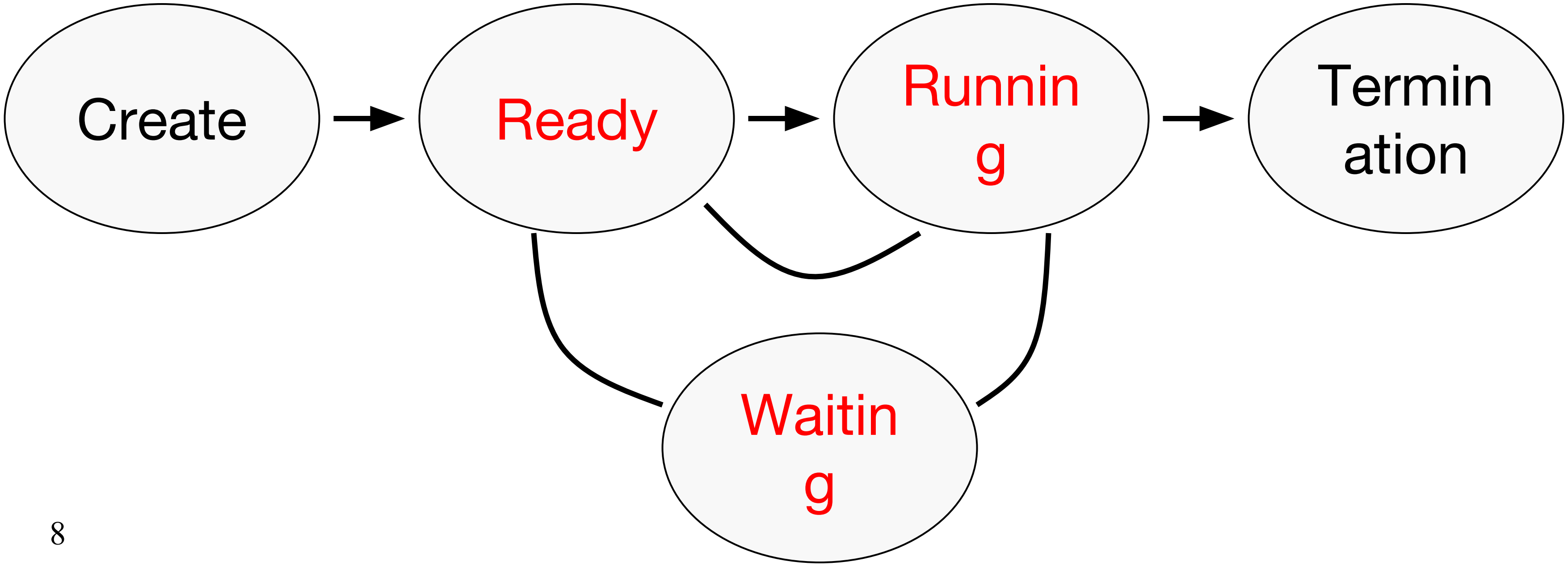
- Which of the following is more important?
  - Process isolation (the illusion that each process is the only one on the machine).
  - Process interaction (synchronization, inter-process communication).

Program vs. Process

- Program
  - Executable code
  - No dynamic state
- Process
  - An instance of a program in execution
  - With its own control flow (illusion of a processor)
  - … & private address space (illusion of memory)
  - State including code, data, stack, registers, instruction pointer, open file descriptors, …
  - Either running, waiting, or ready…
- Can run multiple instances of the same program

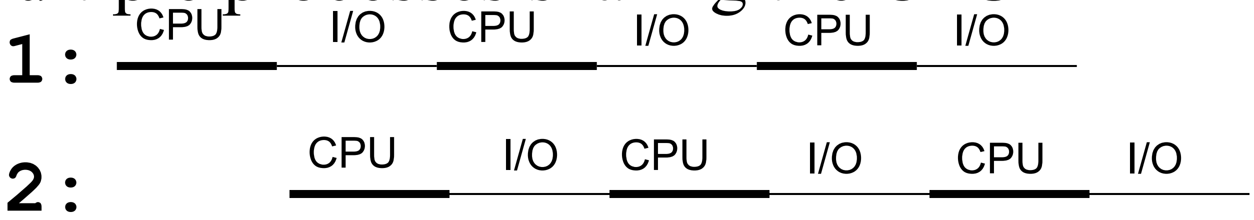
Life Cycle of a Process

- Running: instructions are being executed
- Waiting: waiting for some event (e.g., I/O finish)
- Ready: ready to be assigned to a processor



Many Processes Running “Concurrently”

- Multiple processes sharing the CPU



1: CPU I/O CPU I/O CPU I/O

2: CPU I/O CPU I/O CPU I/O

- Processor switches context between the two
  - When process blocks waiting for operation to complete
  - When process finishing using its share of the CPU
- But, how do multiple processes start running
  - How are they invoked in the first place?

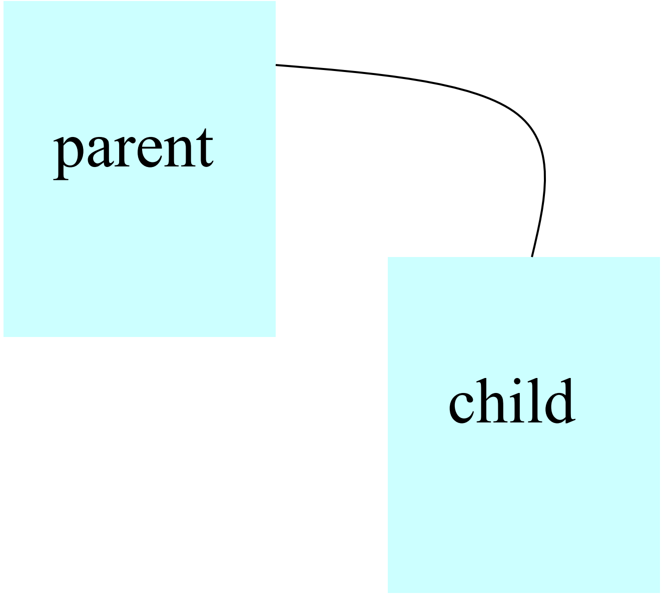
Why Start a New Process?

- Run a new program
  - E.g., shell executing a program entered at command line
  - Or, even running an entire pipeline of commands
  - Such as "**`wc –l * | sort | uniq -c | sort –nr`**"
- Run a new thread of control for the same program
  - E.g. a Web server handling a new Web request
  - While continuing to allow more requests to arrive
  - Essentially time sharing the computer

Fork System Call

- Create a new process
  - Child process inherits state from parent process
  - Parent and child have separate copies of that state
  - Parent and child share access to any open files

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



Creating a New Process - fork()

```
pid = fork();

if (pid == -1) {
   fprintf(stderr, "fork failed\n");
   exit(1);
}

if (pid == 0) {
   printf(“This is the child\n");
   exit(0);
}

if (pid > 0) {
   printf(“This is parent. The child is %d\n", pid);
   exit(0);
}
```

Points to Note

- fork() is called once …
- … but it returns twice!!
  - Once in the parent and
  - Once in the child
  - See example1.c
- Fork() basically duplicates the parent process image
  - Both processes are exactly the same after the fork() call.
    - Are there any dependence between the two processes?
  - Provide a way to distinguish the parent and the child.

Points to Note

- How to distinguish parent and child??
  - Return value in child = 0
  - Return value in parent = process id of child
  - See example2.c
- What about the data in the program?
  - See example6.c.
- Return value of -1 indicates error in all UNIX system calls – another UNIX convention
- Is it true: All processes are created by fork() in UNIX?

Fork System Call

- Fork is called once
  - But returns twice, once in each process
- Telling which process is which
  - Parent: fork() returns the child's process ID
  - Child: fork() returns a 0

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

Example: What Output?

```
int main()
{
   pid_t pid;
   int x = 1;

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

Fork

- Inherited:
  - user and group IDs
  - signal handling settings
  - stdio
  - file pointers
  - current working directory
  - root directory
  - file mode creation mask
  - resource limits
  - controlling terminal
  - all machine register states
  - control register(s)
  - . . .

- Separate in child
  - process ID
  - address space (memory)
  - file descriptors
  - parent process ID
  - pending signals
  - timer signal reset times
  - . . .

Wait

- Parent waits for a child (system call)
  - blocks until a child terminates
  - returns pid of the child process
  - returns –1 if no children exists (already exited)
  - `status`

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t wait(int *status);
```

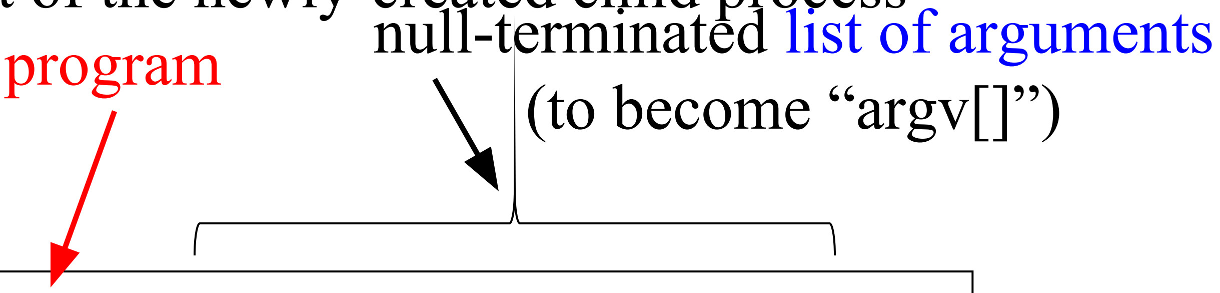
- Parent waits for a specific child to terminate

```
#include <sys/types.h>
#include <sys/wait.h>

pid_t waitpid(pid_t pid, int *status, int options);
```

Executing a New Program

- Fork copies the state of the parent process
  - Child continues running the parent program
  - … with a copy of the process memory and registers
- Need a way to invoke a new program
  - In the context of the newly-created child process
- Example



program

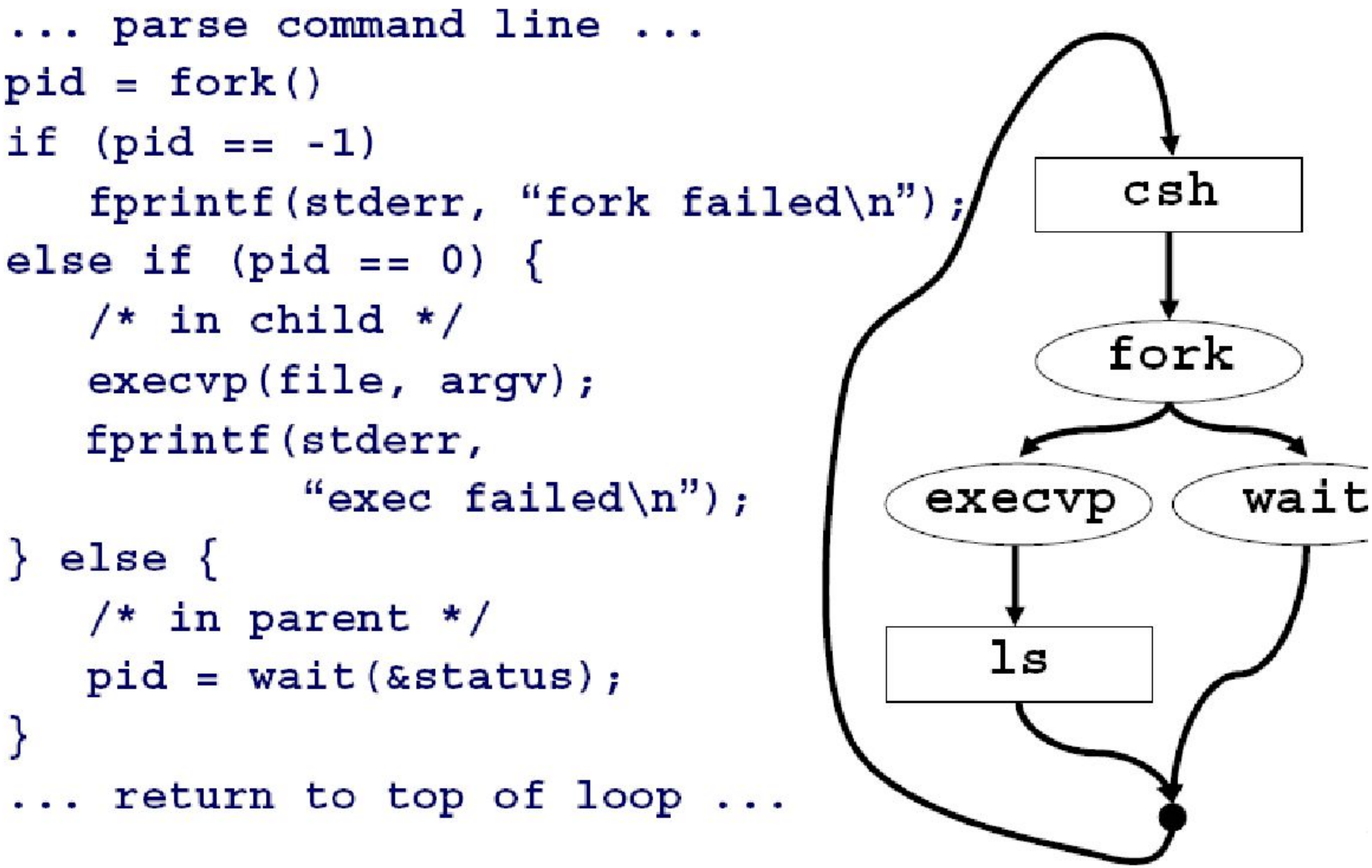
null-terminated list of arguments

(to become “argv[]”)

```
execlp(“ls”, “ls”, “-l”, NULL);
fprintf(stderr, “exec failed\n”);
exit(1);
```

Combining Fork() and Exec()

- Commonly used together by the shell



```
... parse command line ...
pid = fork()
if (pid == -1)
   fprintf(stderr, "fork failed\n");
else if (pid == 0) {
   /* in child */
   execvp(file, argv);
   fprintf(stderr,
           "exec failed\n");
} else {
   /* in parent */
   pid = wait(&status);
}
... return to top of loop ...
```

System

- Convenient way to invoke fork/exec/wait
  - Forks new process
  - Execs command
  - Waits until it is complete

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

- Example:

```
int main()
{
        system(“echo Hello world”);
}
```

Examining Processes in Unix

- *ps command*
  - Standard process attributes
- /proc directory
  - More interesting information.
  - Try “man proc”
- *Top, vmstat command*
  - *Examining CPU and memory usage statistics.*

Running an existing command in a program – exec()

- `int execl(char * pathname, char * arg0, … , (char *)0);`

- `int execv(char * pathname, char * argv[]);`

- `int execle(char * pathname, char * arg0, … , (char *)0, char envp[]);`

- `int execve(char * pathname, char * argv[], char envp[]);`

- `int execlp(char * filename, char * arg0, … , (char *)0);`

- `int execvp(char * filename, char * argv[]);`

execv

- `int execv(char * pathname, char * argv[]);`

Example: to run “/bin/ls –l –a /”

pathname: file path for the executable

char *argv[]: must be exactly the same as the C/C++ command line argument. E.g argv[4] must be NULL.

See example3d.c

Properties of `exec()`

- Replaces current process image with new program image.
  - E.g. parent image replaced by the new program image.
  - If successful, everything after the exec() call will NOT be executed.
    - Will execv() return anything other than -1?

Terminating a process

- exit (int status)
  - Clean up the process (e.g close all files)
  - Tell its parent processes that he is dying (SIGCHLD)
  - Tell child processes that he is dying (SIGHUP)
  - Exit status can be accessed by the parent process.
- When a process exits – not all resources associated with the process are freed yet!!
  - ps can still see the process (<defunct>), see example6.c

Parent/child synchronization

- Parent created the child, he has the responsibility to see it through:
  - check if the child is done.
    - wait, waitpid
      - This will clean up all trace of the child process from the system. See example6.c
  - check if the exit status of the child
    - pid_t wait(int *stat_loc), see example4.c
  - Some others such as whether the child was killed by an signal. etc
- A child has no responsibility for the parent

- Processes are identified by a process id (pid)
  - getpid(): find your own pid
  - getppid(): find the pid of the parent

- See example5.c for the time for system calls versus regular routine calls.

- A question: How to implement the *system* routine?

Simple program to fork a new process

```
#include <stdio.h>
main (int argc, char *argv[])

 { int pid ;
   char *args[2] ;

   printf("Ready to FORK\n") ;
   pid =  fork() ;
   if (pid ==0)

       printf("I AM THE CHILD!!\n") ;

  else
     printf("I AM THE PARENT!!!\n") ;
}
```

Simple program to start a new process executing

```
#include <stdio.h>
main (int argc, char *argv[])

  { int pid ;
    char *args[2] ;

    printf("Ready to FORK\n") ;
    pid =  fork() ;
    if (pid ==0)

       { printf("I AM THE
CHILD!!\n") ;
        args[0] = "./a.out" ;
        args[1] = NULL ;
        execv("./a.out", args) ;
        printf("OPPSSSS\n") ;
       }
```

Simple program to show how children get out of control if not monitored by the parent.

```
#include <stdio.h>
main (int argc, char *argv[])

  { int pid ;
    char *args[2] ;

    printf("Ready to FORK\n") ;
    pid =  fork() ;
    if (pid ==0)

       { printf("I AM THE CHILD!!\n") ;
        args[0] = "./a.out" ;
        args[1] = NULL ;
        execv("./a.out", args) ;
        printf("OPPSSSS\n") ;
       }

  else
      printf("I AM THE PARENT!!!\n") ;
  //wait(NULL) ;
}
```