NWEN 241
Getting closer to the system
Process Management

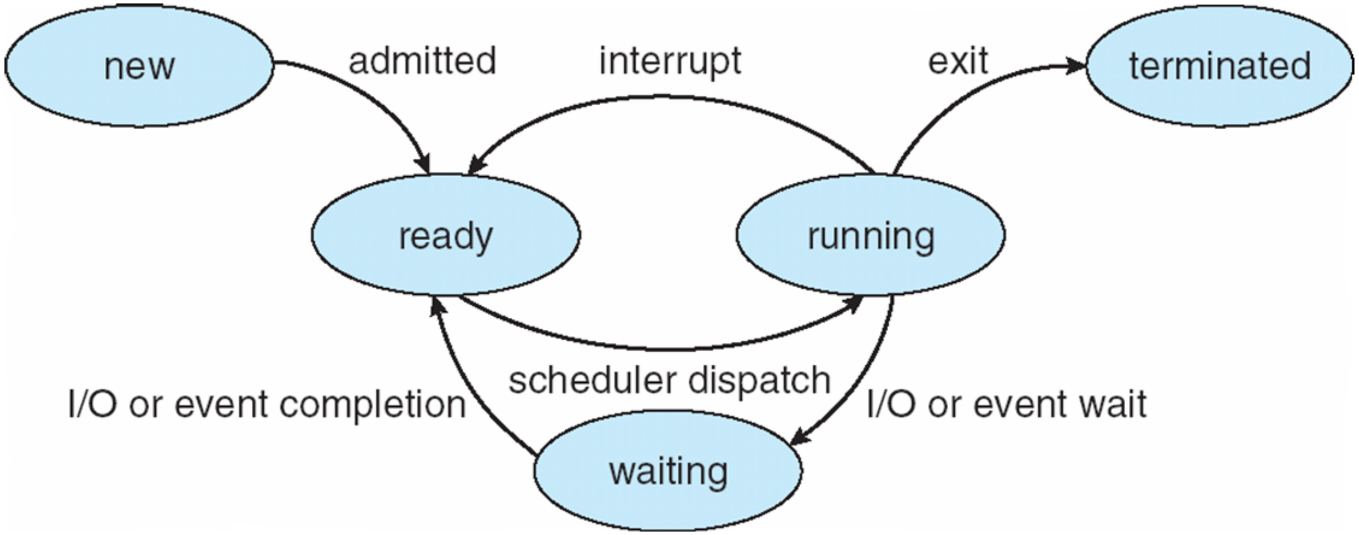
Winston Seah

School of Engineering and Computer Science
Victoria University of Wellington

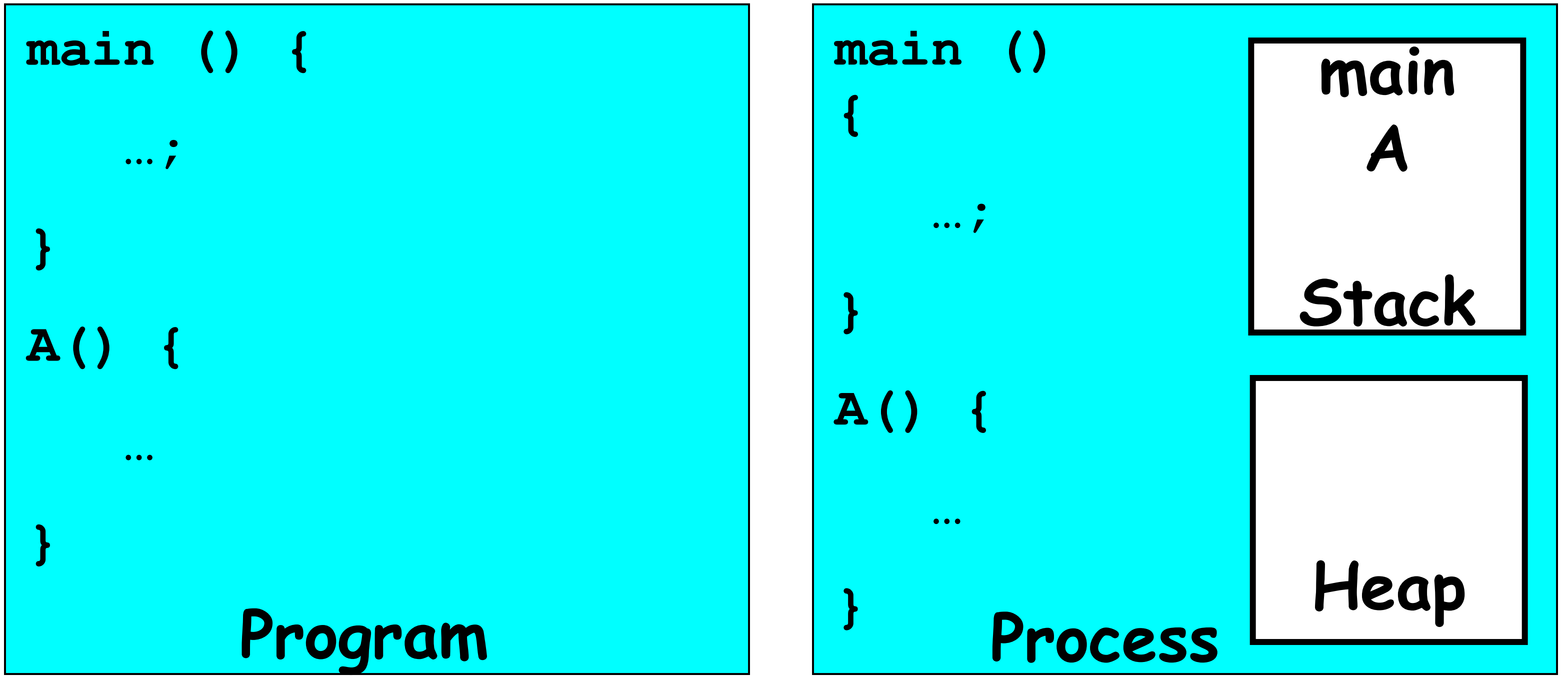
Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

Process vs Program

- Process – a program in execution
- A process includes (among other things):
  - program counter
  - data section
  - stack
- Process life-cycle



Process vs Program



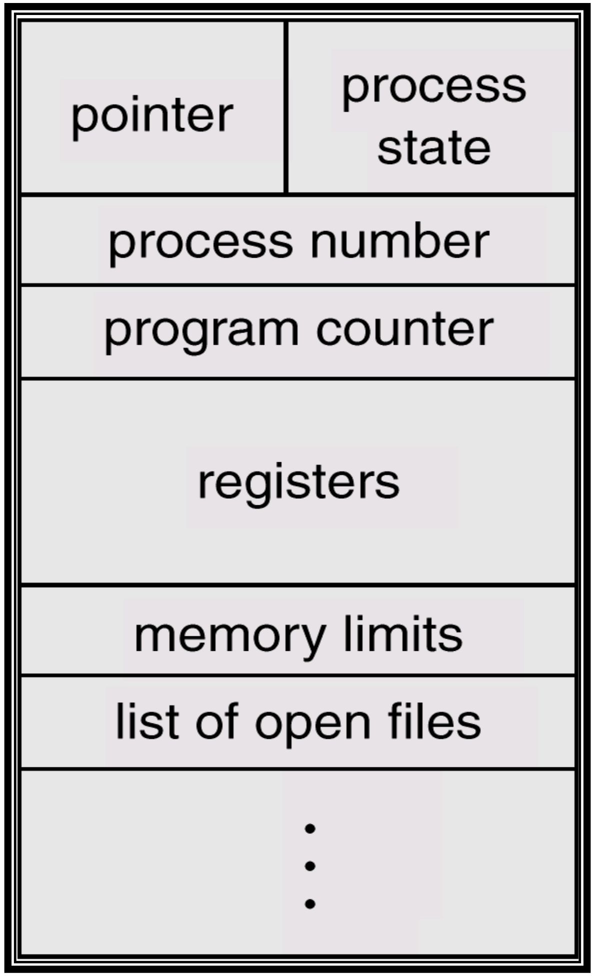
```
main () {
    …;
}
A() {
    …
}
```

```
main ()
{
    …;
}
A() {
    …
}
```

- Program is static, with the potential for execution
- Process is a program in execution and have a state
- One program can be executed several times and thus has several processes

Process Control Block (PCB)

- Information associated with each process
  - Process state
  - Program counter
  - CPU registers
  - CPU scheduling information
  - Memory-management information
  - Accounting information
  - I/O status information
- A process is named using its process ID (PID) or process #
- Data is stored in a process control block (PCB)



pointer

process state

process number

program counter

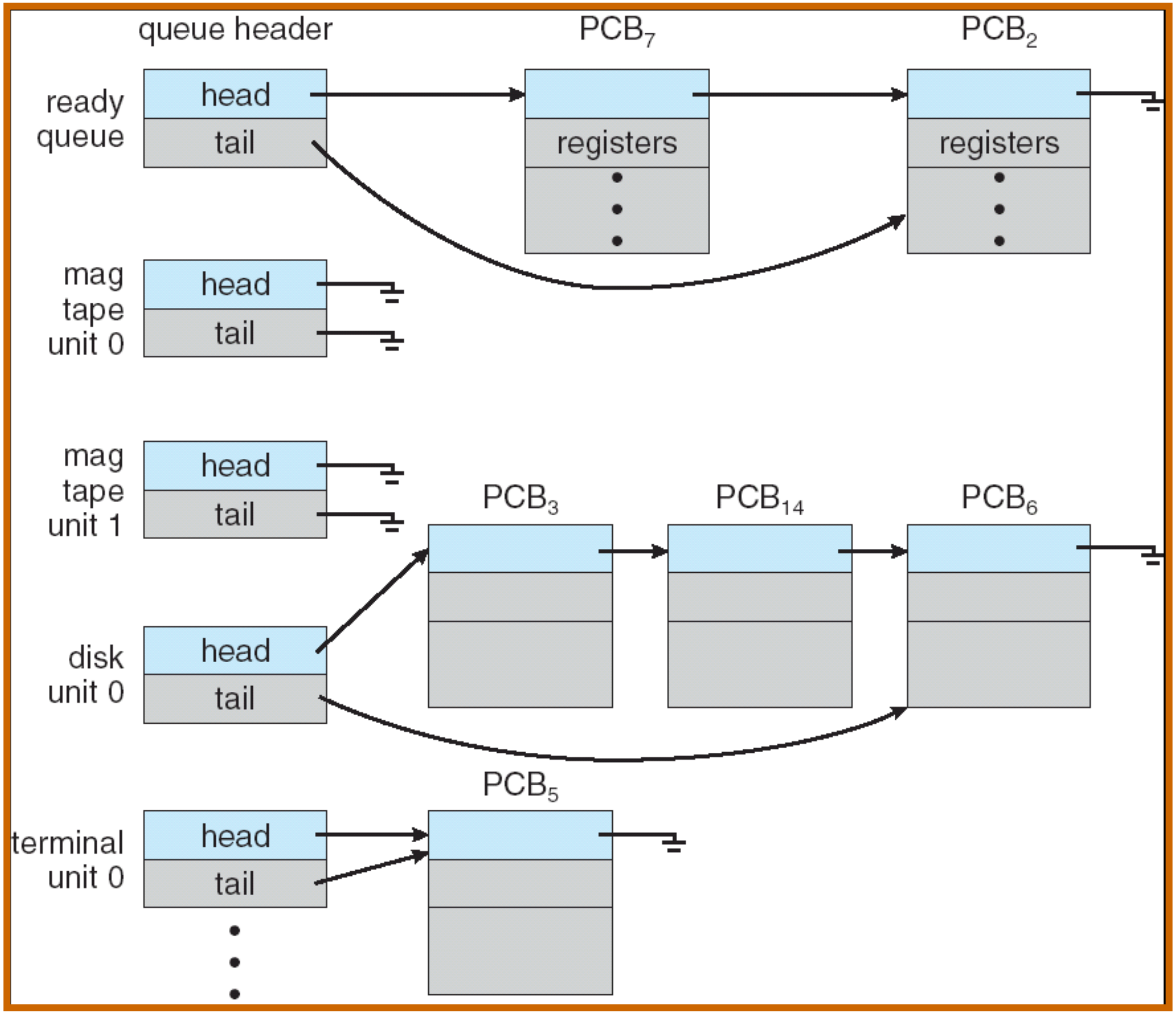
registers

memory limits

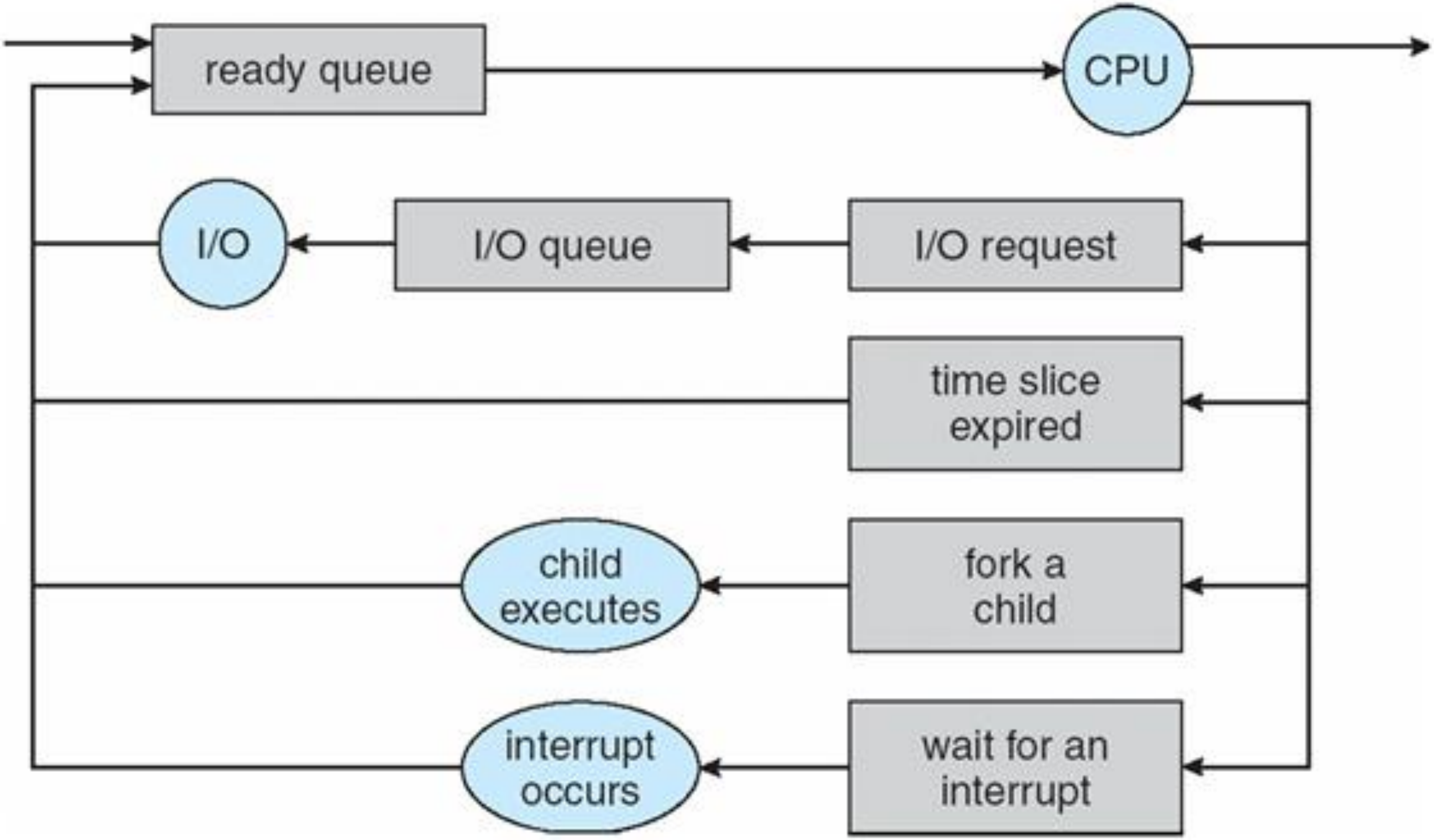
list of open files

⋮

Ready Queue and I/O Device Queues



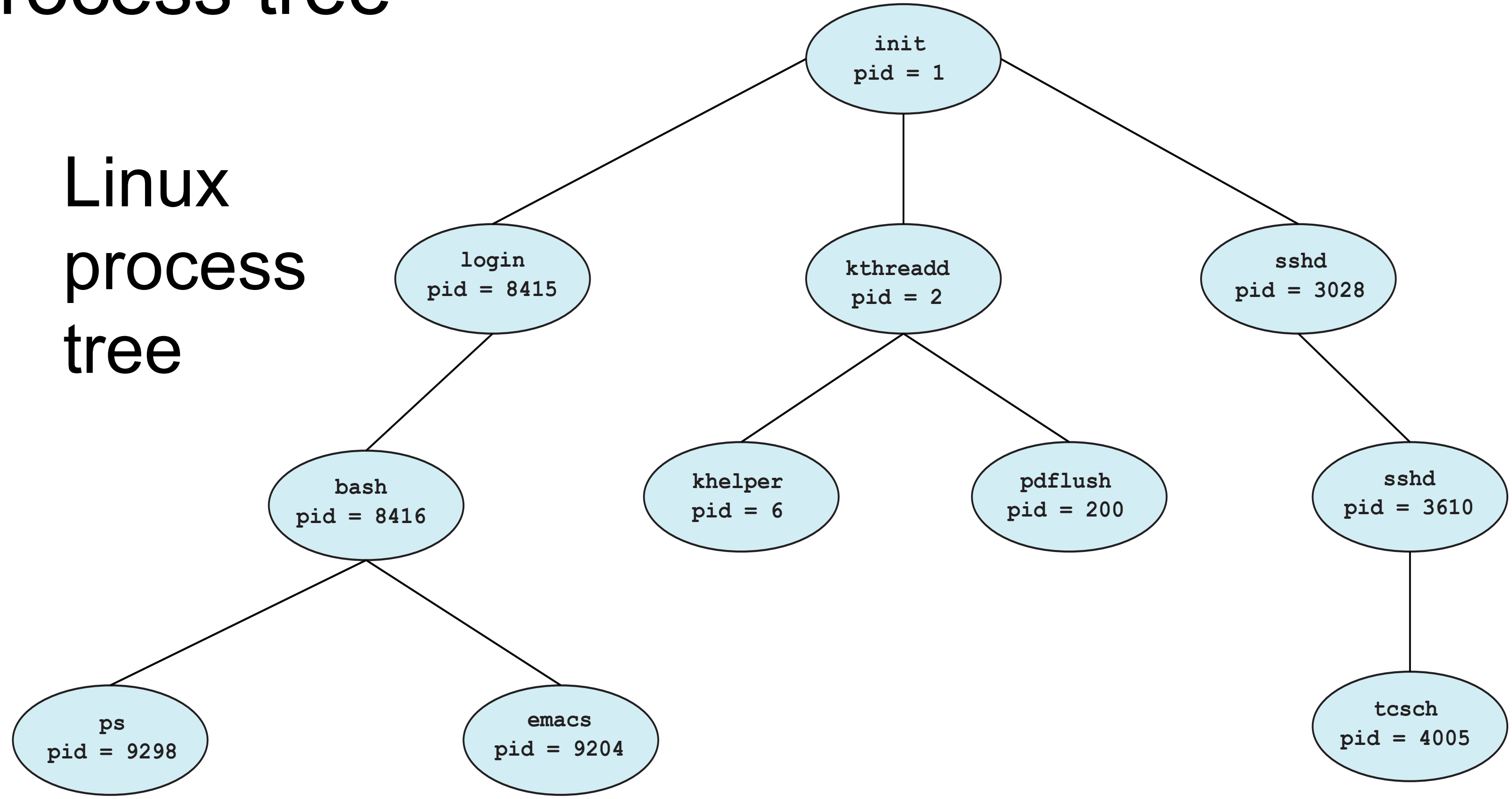
Process Scheduling / Switching



Process Management

- A process is created by another process, which, in turn create other processes → process tree

Linux process tree

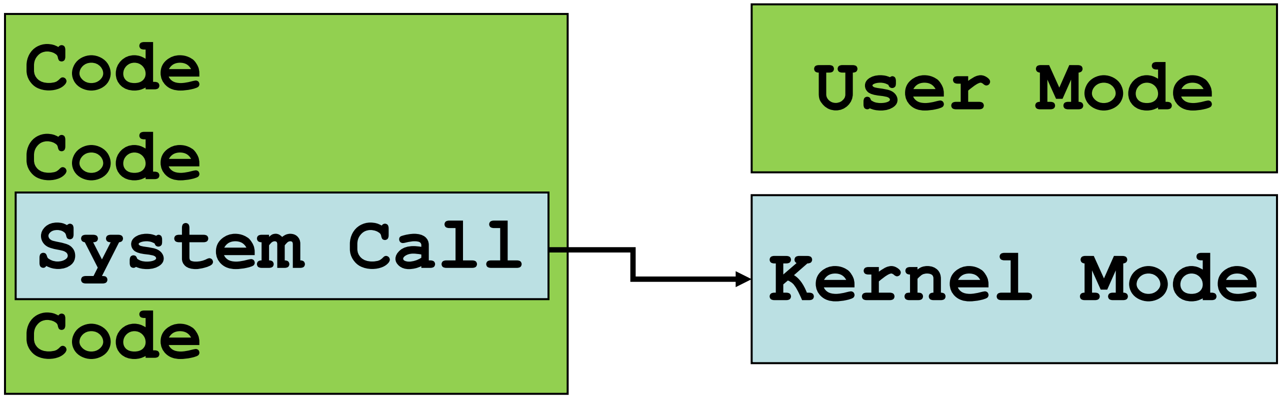


Process Management

- Parent and child process
  - In Linux, using “ps –f”, the PPID field is the parent
  - The first process is ***init*** having process ID **1**
- After creating a child, the parent may either wait for it to finish or continue concurrently
- Process Management in C using System Calls
  - `fork()`
  - `exec()`
  - `wait()`
  - `exit()`

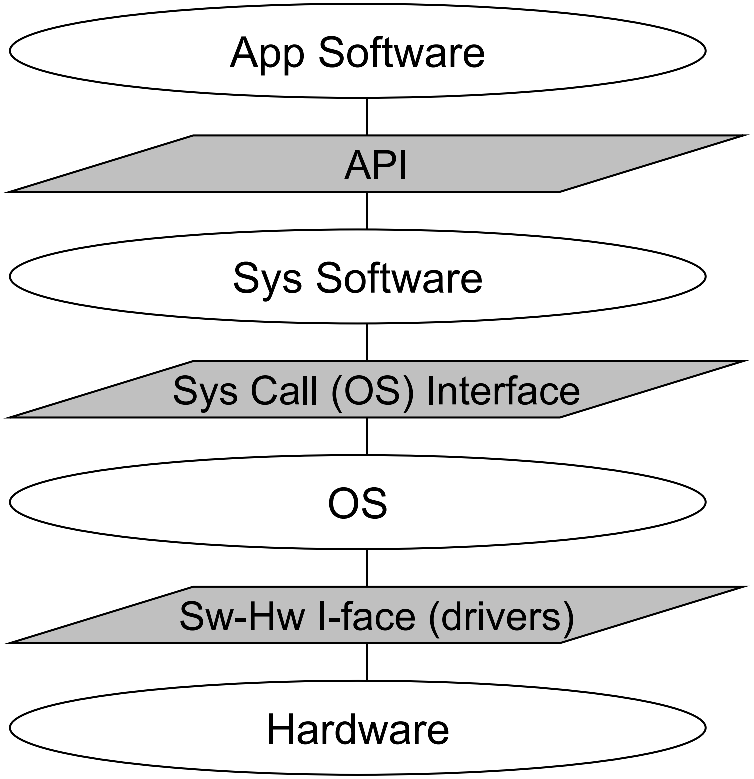
System Call

- A direct request to the operating system to do something on behalf of the program
- Typically programs are executed in ***user*** mode
- System call allows a switch from ***user*** mode to ***kernel*** mode



- The ***kernel*** is the core of the operating system for managing ***processes***, ***files***, ***networking***, etc..

System Call Interface



App Software

API

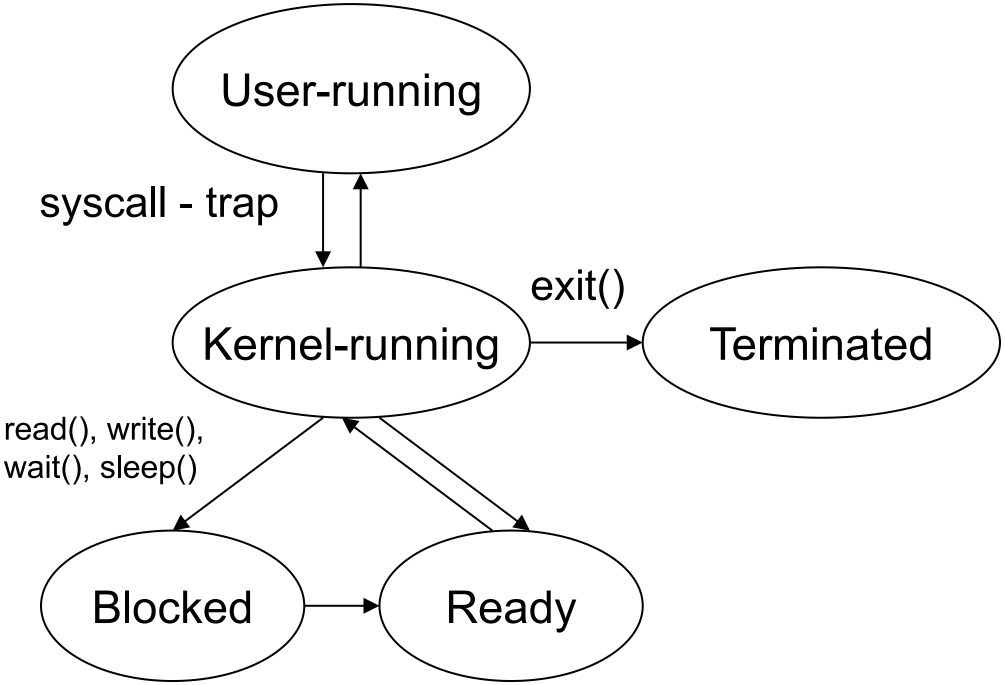
Sys Software

Sys Call (OS) Interface

OS

Sw-Hw I-face (drivers)

Hardware



User-running

syscall - trap

Kernel-running

exit()

Terminated

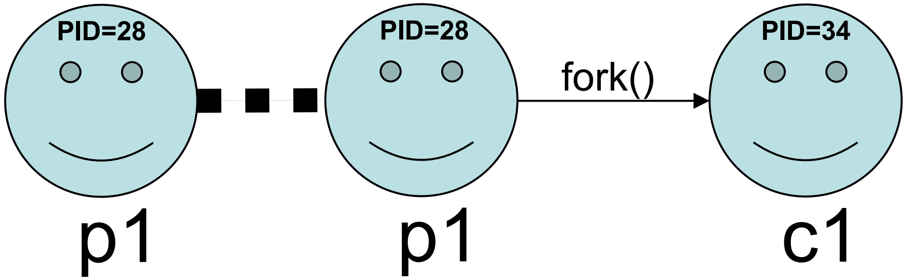
read(), write(),
wait(), sleep()

Blocked

Ready

Process Creation with `fork()` System Call

- A process calling **`fork()`** spawns a child process.
- After a successful **`fork()`** call, two copies of the original code will be running.
  - Parent process – *return value* of **`fork()`** → *child PID*.
  - New child process – *return value* of **`fork()`** → 0.
- **`fork()`** is called once, but returns twice!
- After **`fork()`** both the parent and the child are executing the same program.
- On error, **`fork()`** returns -1.



Consider a piece of program (see examples latex):

```
...
pid_t pid = fork();
printf(“PID: %d\n”, pid);
...
```

The parent will print:

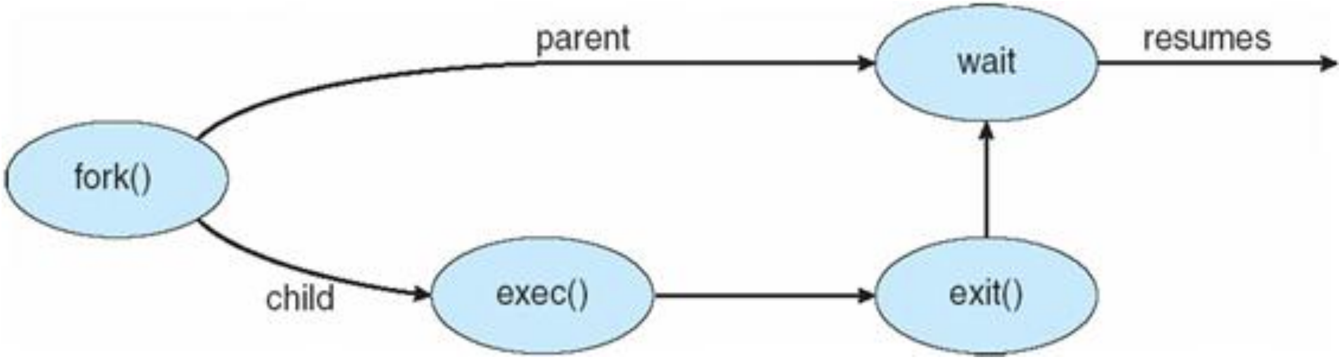
```
PID: 34
```

And the child will **always** print:

```
PID: 0
```

exec() System call (1)

- The **exec()** call replaces a current process' image with a new one (i.e. loads a new program within current process).
- Upon success, exec() **never** returns to the caller. If it does return, it means the call failed. Typical reasons are: non-existent file (bad path) or bad permissions.
- Arguments passed via **exec()** appear in the **argv[]** of the **main()** function.



`exec()` System call (2)

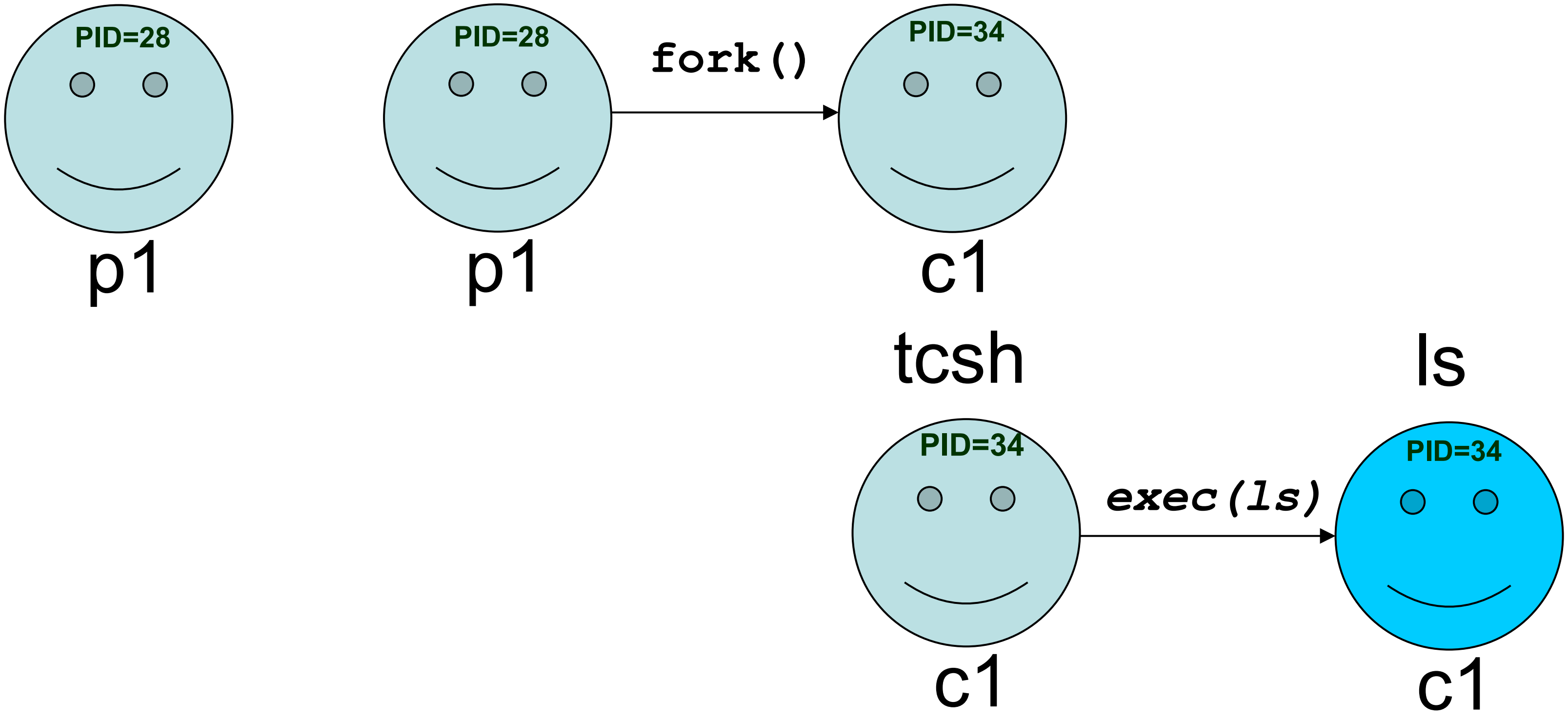
- There is no system call specifically by the name **`exec()`**. By **`exec()`** we usually refer to a family of calls:
  - `int execl(char *path, char *arg, ...);`
  - `int execv(char *path, char *argv[]);`

  -
    ```
    int execle(char *path, char *arg, ...,
                 char *envp[]);
    ```

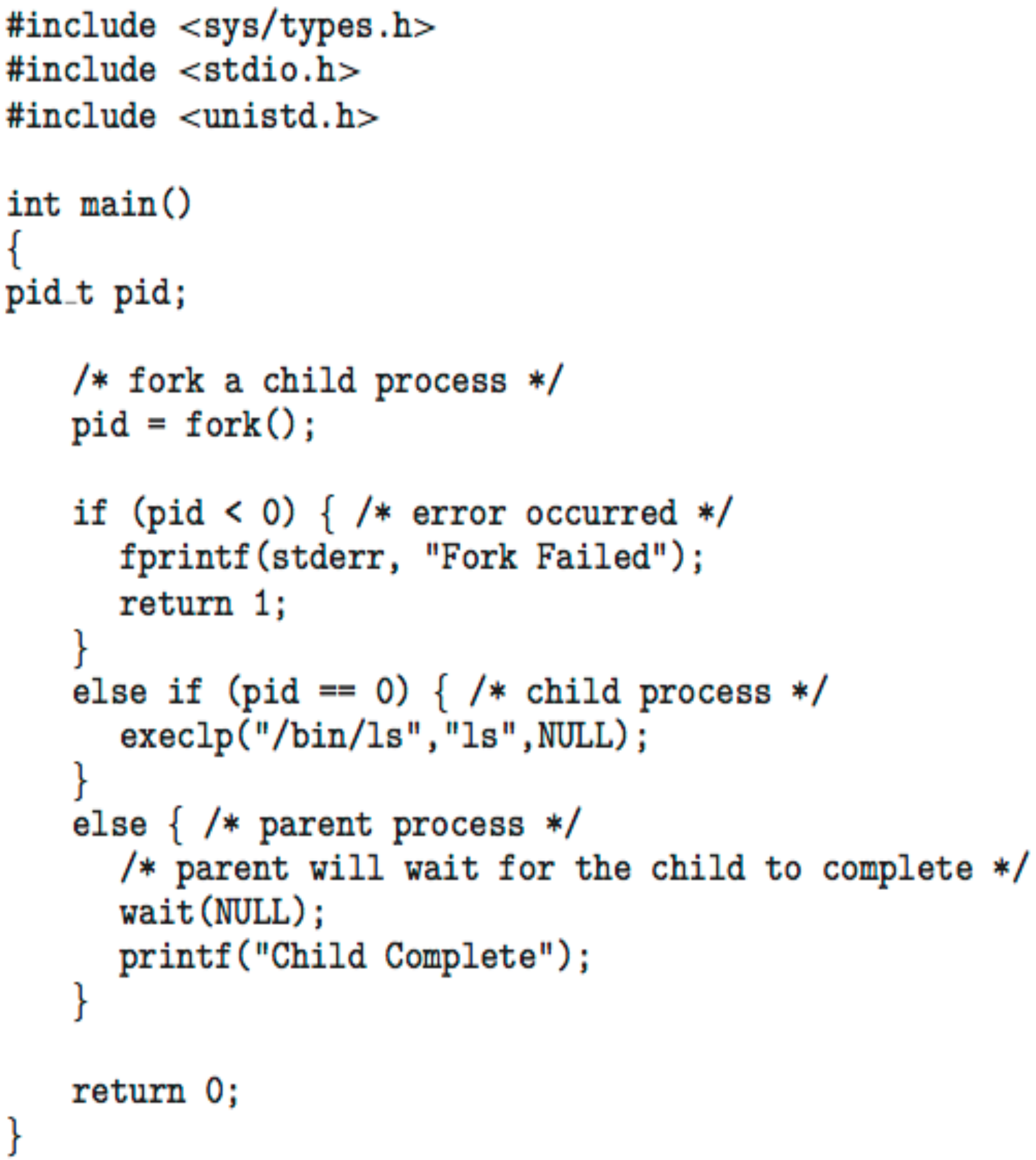
  -
    ```
    int execve(char *path, char *argv[],
                 char *envp[]);
    ```
  - `int execlp(char *file, char *arg, ...);`
  - `int execvp(char *file, char *argv[]);`
- The various options *l*, *v*, *e*, and *p* mean:
  - ***l*** : an argument list,
  - ***v*** : an argument vector,
  - ***e*** : an environment vector, and
  - ***p*** : a search path.

`fork()` and `exec()` together

- Often after doing **`fork()`** we want to load a new program into the child.
- Most common e.g. a shell.



Example of forking separate processes



```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

    /* fork a child process */
    pid = fork();

    if (pid < 0) { /* error occurred */
       fprintf(stderr, "Fork Failed");
       return 1;
    }
    else if (pid == 0) { /* child process */
       execlp("/bin/ls","ls",NULL);
    }
    else { /* parent process */
       /* parent will wait for the child to complete */
       wait(NULL);
       printf("Child Complete");
    }

    return 0;
}
```

`wait()` Call System

- Forces the parent to suspend execution, i.e. wait for its children or a specific child to die (*terminate* is more appropriate terminology, but a bit less common).

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

- **`waitpid()`** waits for the child with specific PID.

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

- The status, if not NULL, stores exit information of the child, which can be analyzed by the parent.
- The return value is:
  - PID of the exited process, if no error
  - (-1) if an error has happened

`exit()` System Call

- Gracefully terminates process execution, meaning it does clean up and release of resources, and puts the process into the ***zombie*** state → *terminated but still waiting for parent process to read its exit status*.
- By calling **`wait()`**, the parent cleans up all its zombie children.
- **`exit()`** specifies a return value from the program, which a parent process might want to examine as well as status of the dead process.
- **`_exit()`** call is another possibility of quick death without cleanup.

Example of `wait()` and `exit()`

```c
#include <stdio.h>
#include <stdlib.h>

main()
{
  int pid; int rv;

  pid=fork();
  switch(pid){
    case -1:
      printf("Error -- Something went wrong with fork()\n");
      exit(1); // parent exits
    case 0:
      printf("CHILD: This is the child process!\n");
      printf("CHILD: My PID is %d\n", getpid());
      printf("CHILD: My parent's PID is %d\n", getppid());
      printf("CHILD: Enter my exit status: ");
      scanf(" %d", &rv);
      printf("CHILD: I'm outta here!\n");
      exit(rv);
    default:
      printf("PARENT: This is the parent process!\n");
      printf("PARENT: My PID is %d\n", getpid());
      printf("PARENT: My child's PID is %d\n", pid);
      printf("PARENT: I'm now waiting for my child to exit()...\n");
      wait(&rv);
      printf("PARENT: I'm outta here!\n");
  }
}
```

More about wait() and exit()

- Should not interpret the status value of system call **wait(&status)** literally. If **&status** is not NULL, **wait()** stores status information in the **int** to which it points.
- Value returned by **exit(&status)** is moved to 2nd byte and 1st (lowest) byte is used to store the status information.
- In previous example:

```
   scanf(" %d", &rv); // if value of x is entered
   ...
   exit(rv);
```

0	0	0	x

```
...
   wait(&rv); // the rv contents will be x left shift by
              // 8 bits and additional status written into
              // lowest 8 bit
```

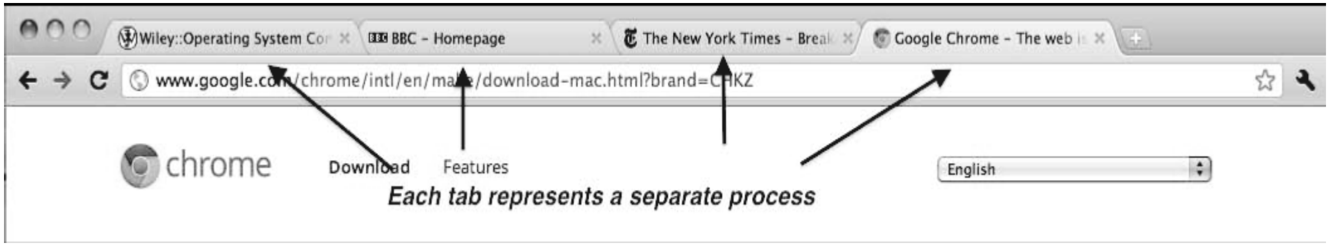
0	0	x	status

More about `wait()` and `exit()`

- This **`status`** integer can be inspected with macros (which take the integer itself as an argument, **not** a pointer to it):
  - **`WIFEXITED(status)`**
  - **`WEXITSTATUS(status)`** This macro should be employed only if **`WIFEXITED`** returned true.
  - **`WIFSIGNALED(status)`**
  - **`WTERMSIG(status)`** This macro should be employed only if **`WIFSIGNALED`** returned true.
  - **`WCOREDUMP(status)`**
  - **`WIFSTOPPED(status)`**
  - **`WSTOPSIG(status)`**
  - **`WIFCONTINUED(status)`**

Multiprocessing – Google Chrome

- Many web browsers ran (in the past) as single process (some still do) → If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multi-process with 3 different types of processes:
  - **Browser** process manages UI, disk and network I/O
  - **Renderer** process renders web pages, deals with HTML, Javascript. A new renderer created for each website opened → Runs in **sandbox** restricting disk and network I/O, minimizing effect of security exploits
  - **Plug-in** process for each type of plug-in


Wiley::Operating System Co
BBC - Homepage
The New York Times - Break
Google Chrome - The web
www.google.com/chrome/intl/en/make/download-mac.html?brand=CHKZ
chrome
Download
Features
English

Each tab represents a separate process