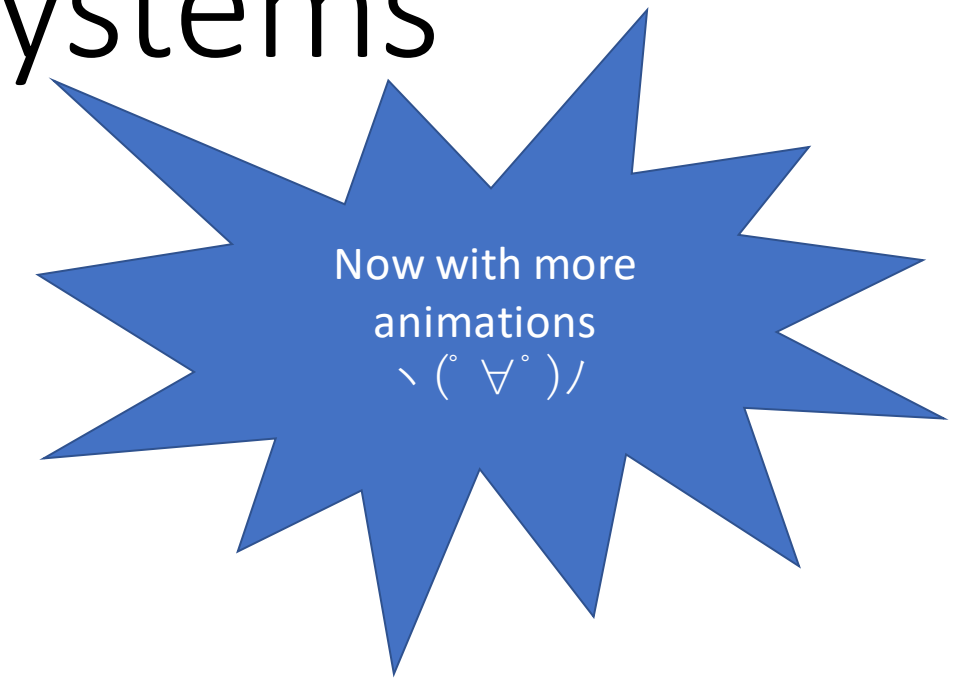
Computer Systems

Exercise 2



Process

- Instance of a program
- Ingredients of a process:
  - Virtual processor (address space, registers, program counter, instruction pointer)
  - Program text (code)
  - Program data (heap, stack)
  - OS stuff (open files, sockets, CPU shares, security rights, process ID (PID))
  - Process ID (PID)
- Purpose of process concept
  - Multitasking
  - Isolation
- Software = running processes + kernel

Dispatching and Scheduling

- The OS is responsible for allocating CPU time to processes
- It does this by selecting a runnable process and running it for some time
- Running a process is called dispatching

- Selecting which process to run is called scheduling – don't worry about that for now

Process creation – spawning new process

Did it work?

```
BOOL CreateProcess(
  in_opt        LPCTSTR              ApplicationName,
  inout_opt     LPTSTR               CommandLine,
  in_opt        LPSECURITY_ATTRIBUTES ProcessAttributes,
  in_opt        LPSECURITY_ATTRIBUTES ThreadAttributes,
  in            BOOL                 InheritHandles,
  in            DWORD                CreationFlags,
  in_opt        LPVOID               Environment,
  in_opt        LPCTSTR              CurrentDirectory,
  in            LPSTARTUPINFO        StartupInfo,
  out           LPPROCESS_INFORMATION ProcessInformation
);
```

address of binary

What to run?

e.g. rights

What rights will it have?

of the calling process (e.g. open files)

What will it see when it starts up?

name, ID

The result

Moral: the parameter space is large!

Process creation – fork() and exec()

- Simplifies creating processes:
  - Fork creates a copy of the current process, same only that child has return value of fork() = 0 and parent has return value of fork() = PID of child
  - Exec() replaces text of calling process with new program.

→ Creates tree of processes

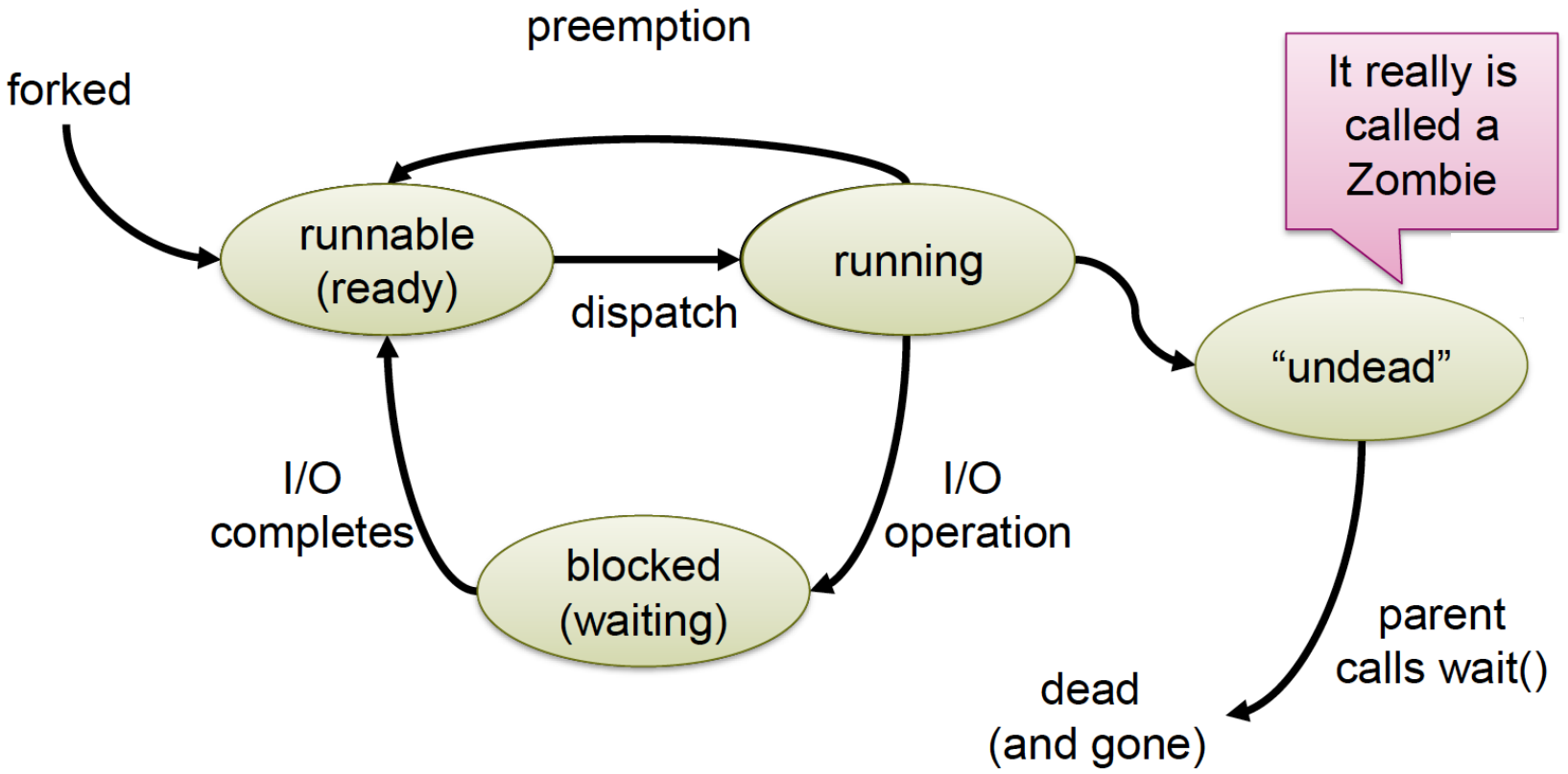
Example fork(), exec()

```
pid_t p = fork();
if ( p < 0 ) {
   // Error…
   exit(-1);
} else if ( p == 0 ) {
   // We're in the child
   execlp("/bin/ls", "ls", NULL);
} else {
   // We're a parent.
   // p is the pid of the child
   wait(NULL);
   exit(0);
}
```

Return code from fork() tells you whether you're in the parent or child (cf. setjmp())

Child process can't actually be cleaned up until parent "waits" for it.

Process lifecycle



Zombies & Orphans

- Why Zombies?
  - If no Zombies, child process could fail and nobody would know because the parent has no chance to catch the exit status (I.e. to call wait() on it)
- What happens with child process whose parents have exited?
  - They are called orphans and get “adopted” by the init process (PID = 1)
  - Init will call wait() on them in order to delete them when they die
- So init is there to reap zombie orphans
- Pro tip: Do not talk about UNIX processes in public

Concurrency without Parallelism

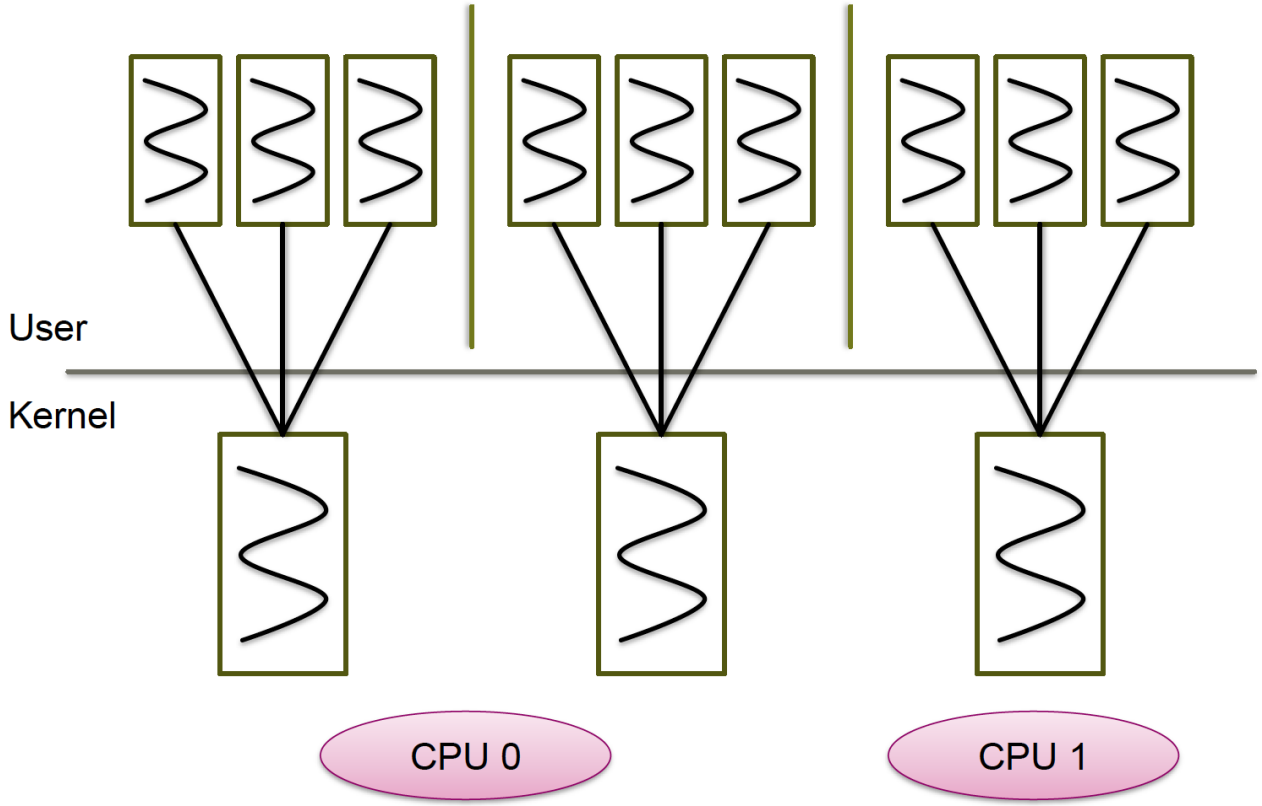
- Example: A system has to react to two buttons (e.g. to manipulate a screen)
- Idea: Just poll the buttons and do stuff
- Problem: This gets hairy fast. What if the buttons have to be polled with different timings?

```
void funWithButtons(){
  If(pollButton1()){
    UI.action1();
  }
  If(button2()){
    UI.action2();
  }
  delay(100);//Poll every
100ms
}
```

Threads

- A thread is part of a process. Multiple threads can exist in one process and share resources such as memory.
- Process – unit of resources, thread – unit of scheduling/execution
- User threads (lightweight processes)
  - Kernel is unaware of them, scheduled in user space
  - Fast to create and manage, but can’t make use of multithreading
- Kernel threads
  - At least on kernel thread exists for each process
  - One kernel thread can be mapped to each logical core and can be swapped once it gets blocked, but takes long to swap

User vs. Kernel Threads



Threads to the rescue!

- Idea: Use Threads
- Problem: This can get hairy too. What if the actions influence some state?

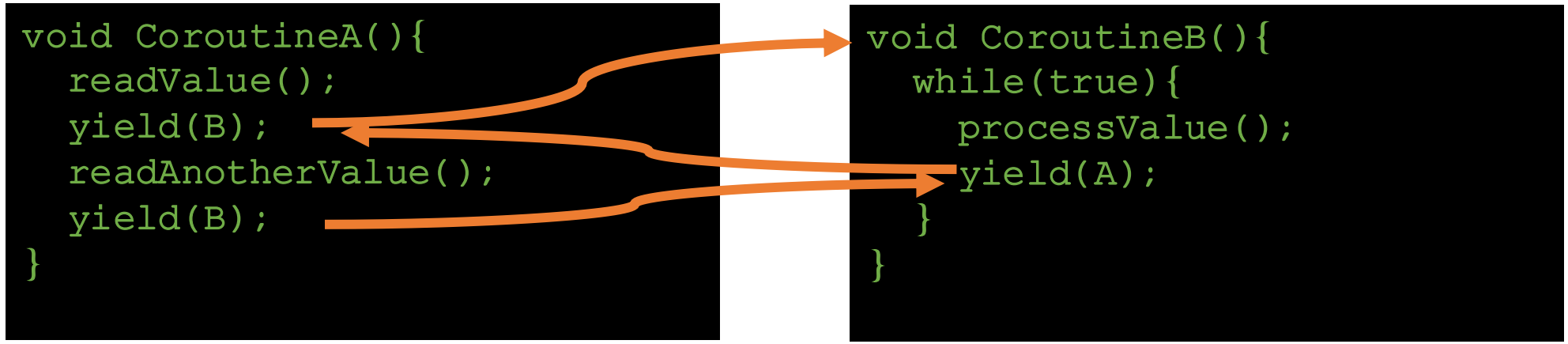
You have a problem.
You decide to use threads.
Now problems two yhou ave.

```
void button1Task(){
  If(pollButton1()){
    UI.action1();
  }
  sleep(100);//Poll every
100ms
}
```

```
void button2Task(){
  If(pollButton2()){
    UI.action2();
  }
  sleep(50);//Poll every 50ms
}
```

Coroutines - Concurrency without parallelism

- "Scheduled" Cooperatively (I.e. a coroutine needs to yield to others)
- Basically two functions jumping between different points of each other
- Another way to look at it: One thread doing the work of two



How do processes (threads) communicate?

- **Shared Memory**
  - Semaphores/Locks
    - Synchronization instructions (CAS, TAS, LL/SC)
  - Transactional Memory

- **Message passing**
  - Asynchronous/Synchronous
  - Blocking/non-blocking
  - Pipes
    - Named/unnamed
  - Upcalls/Signals
    - SIGSEGV from memory management
    - SIGKILL from other process
  - RPC

Message passing

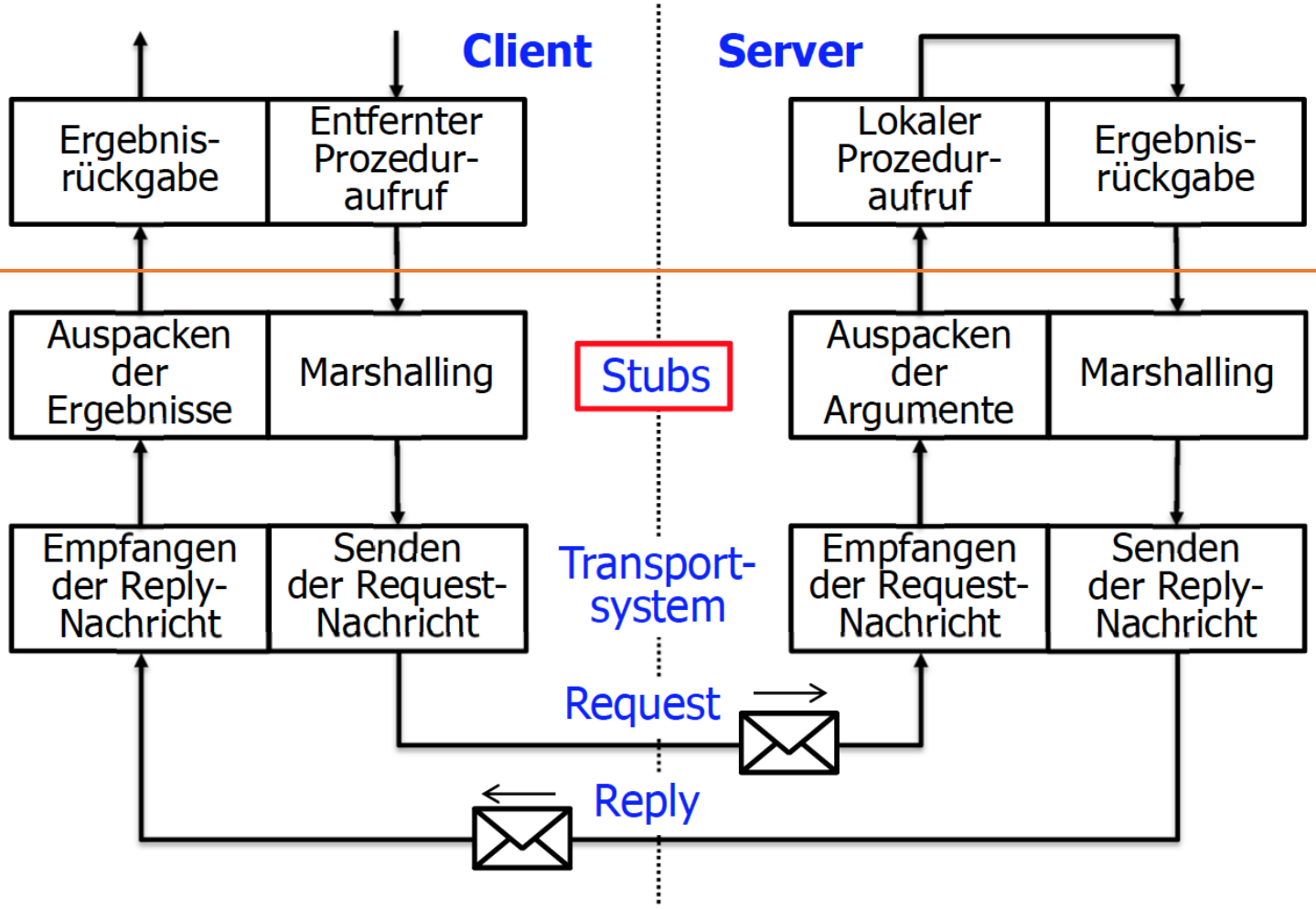
```
void button1Task(){
  If(pollButton1()){
    queue.send(1);
  }
  suspendTask(100);
}
```

```
void button2Task(){
  If(pollButton2()){
    queue.send(2);
  }
  suspendTask(50)
}
```

- Each task takes care of its own state

```
void UITask(){
  while(true){
    switch(queue.receive()){
      case 1: action1();
      case 2: action2();
  }
}
```

RPC


Client
Server
Ergebnis-rückgabe
Entfernter Prozedur-aufruf
Lokaler Prozedur-aufruf
Ergebnis-rückgabe
The magic line
Auspacken der Ergebnisse
Marshalling
Stubs
Auspacken der Argumente
Marshalling
Empfangen der Reply-Nachricht
Senden der Request-Nachricht
Transport-system
Empfangen der Request-Nachricht
Senden der Reply-Nachricht
Request
Reply

Confusing terminology

	Managed By	Scheduling	Shared Memory Space (Generally, except when not)
Process	OS	Preemptive	No
(Kernel) Thread	OS	Preemptive	Yes
User Thread (aka Green Thread)	Application	Preemptive	Yes
Coroutines	Application	Cooperative	Yes

Quiz

- **What is the relation between a process and a program?**
  - A process is a running instance of a program
- **How are processes identified?**
  - By the process ID (PID)
- **In what state is a process after it exited?**
  - Zombie state
- **Are user threads or kernel threads easier to switch?**
  - User space threads
- **How long should you spin for waiting for a lock?**
  - One context switch time
- **What is the point of a name server?**
  - to hold interface references for services