ZHEJIANG UNIVERSITY

Process

Yajin Zhou (http://yajin.org)

Zhejiang University

Acknowledgement: some pages are based on the slides from Zhi Wang(fsu).

Review

- System calls
  - implementation
  - API: wrapper of the system call
  - parameter passing: register, stack, block
- Linking and loading
- OS structure
  - monolithic, micro-kernel, layered, module support, exokernel
- Examples of system calls: fork(), wait(), exec(), ptrace()

Contents

- Process concept
- Process scheduling
- Operations on processes
- Inter-process communication
    - examples of IPC Systems
- Communication in client-server systems

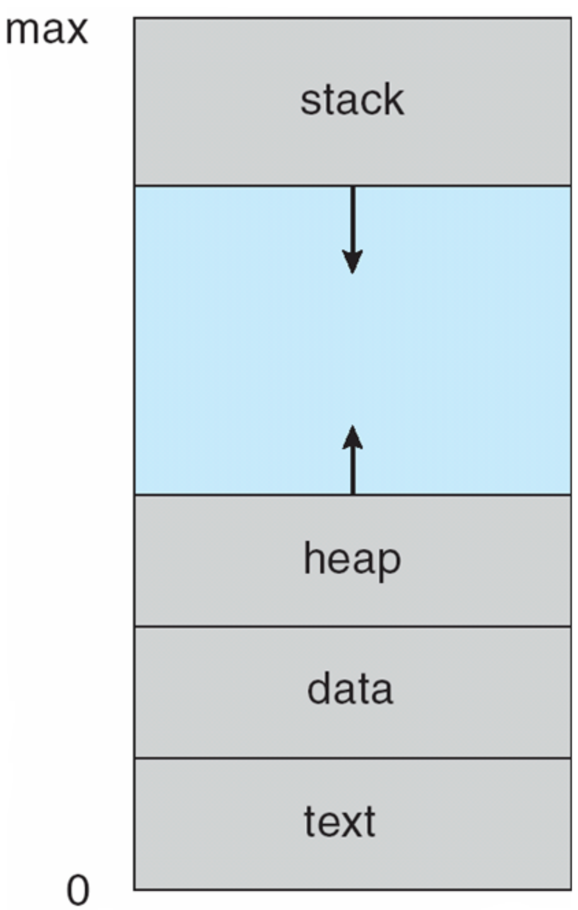
Process Concept

- An operating system executes a variety of programs:
  - **batch system** – jobs
  - **time-shared systems** – user programs or tasks
- Process is **a program in execution**, its execution must progress in sequential fashion
  - a program is static and passive, process is dynamic and active
  - **one program can be several processes** (e.g., **multiple instances** of browser, or **even on instance** of the program)
  - process can be started via GUI or command line entry of its name
    - through system calls

Process Concept

- A process has multiple parts:
  - the program **code**, also called **text section**
  - runtime **CPU states**, including program counter, registers, etc
  - various types of memory:
    - **stack**: temporary data
      - e.g., function parameters, local variables, and *return addresses*
    - **data** section: global variables
    - **heap**: memory dynamically allocated during runtime
      - security: heap feng shui -> how to provide randomness
      - Further reading: FreeGuard: A Faster Secure Heap Allocator (CCS 17), Guarder: A Tunable Secure Allocator (USENIX Sec 18)

Process in Memory

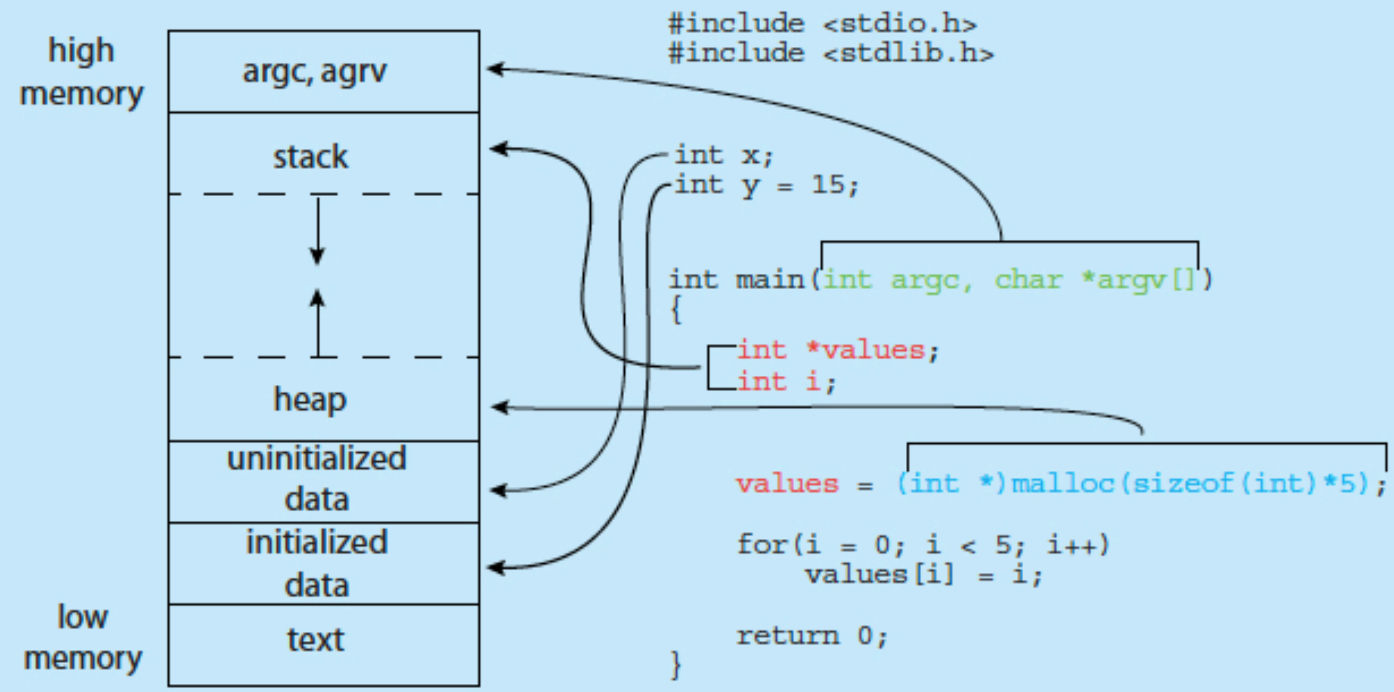



```
os@os:~$ cat /proc/self/maps
00400000-0040c000 r-xp 00000000 08:01 2752536                    /bin/cat
0060b000-0060c000 r--p 0000b000 08:01 2752536                    /bin/cat
0060c000-0060d000 rw-p 0000c000 08:01 2752536                    /bin/cat
0108d000-010ae000 rw-p 00000000 00:00 0                          [heap]
7f3b4c98d000-7f3b4d34c000 r--p 00000000 08:01 3284766            /usr/lib/locale/locale-archive
7f3b4d34c000-7f3b4d50c000 r-xp 00000000 08:01 2102132            /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d50c000-7f3b4d70c000 ---p 001c0000 08:01 2102132            /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d70c000-7f3b4d710000 r--p 001c0000 08:01 2102132            /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d710000-7f3b4d712000 rw-p 001c4000 08:01 2102132            /lib/x86_64-linux-gnu/libc-2.23.so
7f3b4d712000-7f3b4d716000 rw-p 00000000 00:00 0
7f3b4d716000-7f3b4d73c000 r-xp 00000000 08:01 2102104            /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d900000-7f3b4d925000 rw-p 00000000 00:00 0
7f3b4d93b000-7f3b4d93c000 r--p 00025000 08:01 2102104            /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93c000-7f3b4d93d000 rw-p 00026000 08:01 2102104            /lib/x86_64-linux-gnu/ld-2.23.so
7f3b4d93d000-7f3b4d93e000 rw-p 00000000 00:00 0
7ffff3ba3000-7ffff3bc4000 rw-p 00000000 00:00 0                  [stack]
7ffff3bcd000-7ffff3bd0000 r--p 00000000 00:00 0                  [vvar]
7ffff3bd0000-7ffff3bd2000 r-xp 00000000 00:00 0                  [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0          [vsyscall]
```

MEMORY LAYOUT OF A C PROGRAM

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the `argc` and `argv` parameters passed to the `main()` function.



The GNU `size` command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is `memory`, the following is the output generated by entering the command `size memory`:

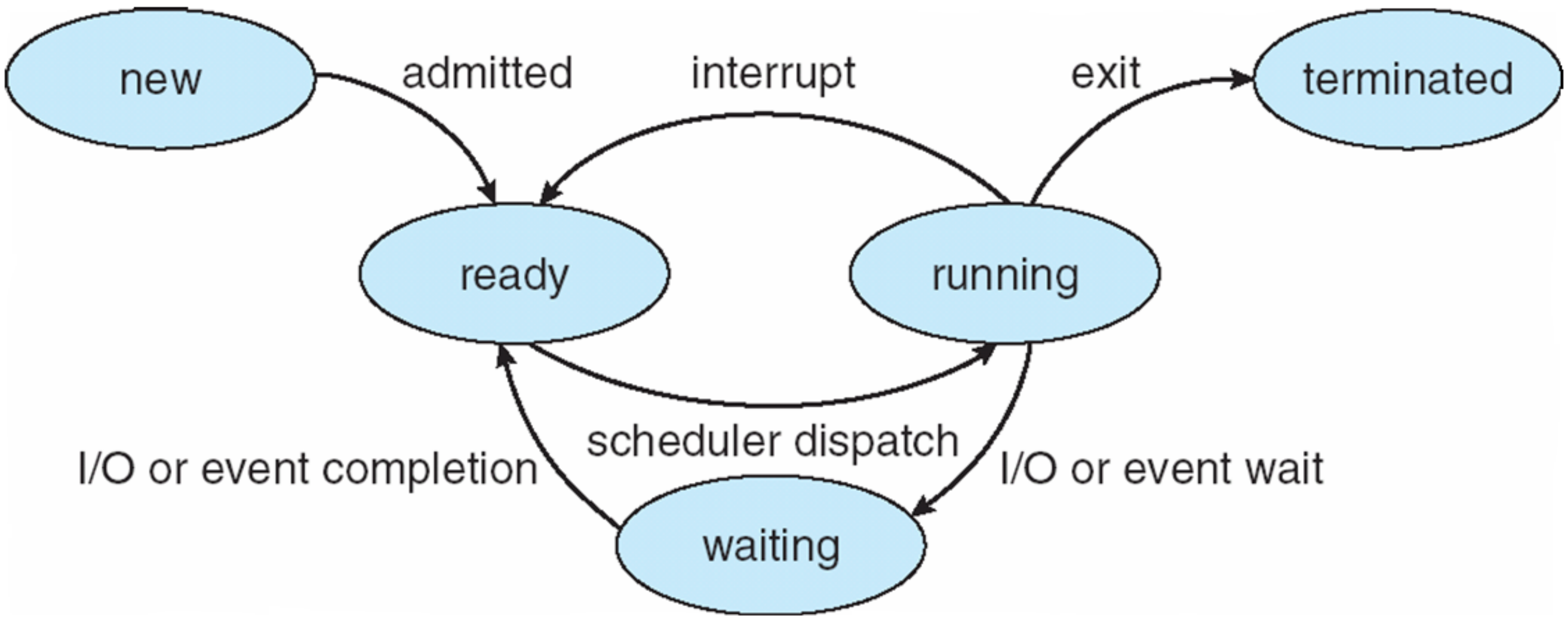
```
text    data    bss    dec     hex    filename
1158    284     8      1450    5aa    memory
```

The `data` field refers to uninitialized data, and `bss` refers to initialized data. (`bss` is a historical term referring to *block started by symbol*.) The `dec` and `hex` values are the sum of the three sections represented in decimal and hexadecimal, respectively.

Process State

- As a process executes, it changes state
  - **new**: the process is being created
  - **running**: instructions are being executed
  - **waiting/blocking**: the process is waiting for some event to occur
  - **ready**: the process is waiting to be assigned to a processor
  - **terminated**: the process has finished execution

Diagram of Process State



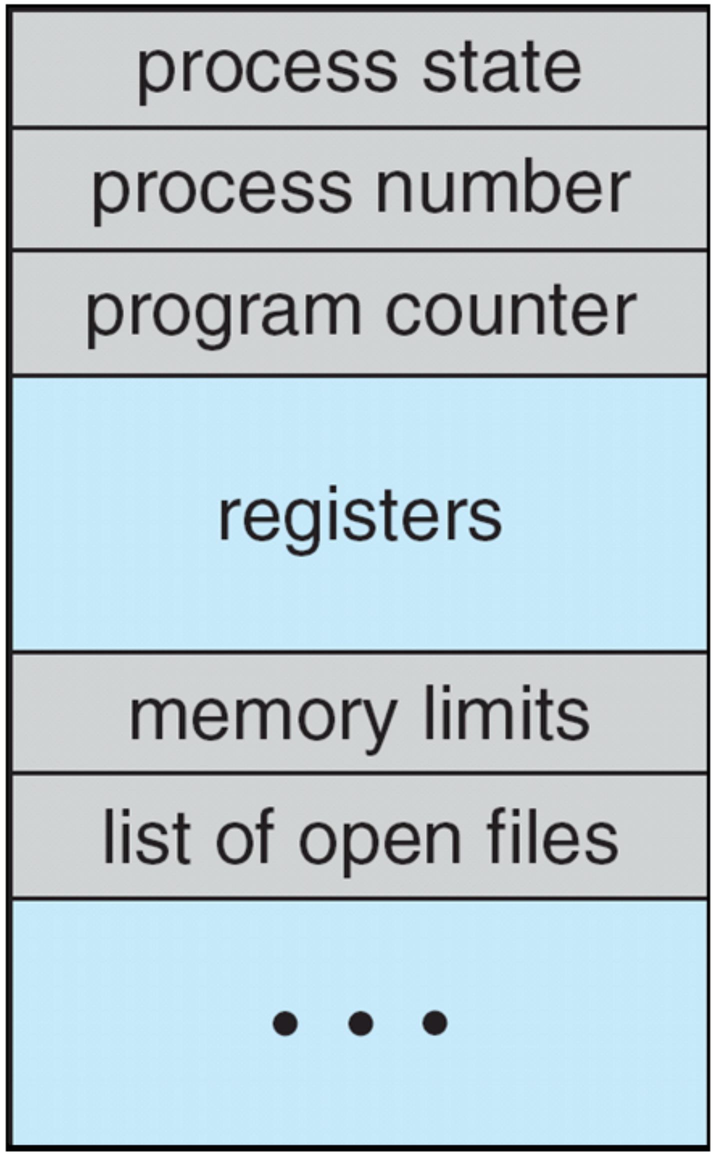
Process State

Time	$Process_0$	$Process_1$	Notes
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	$Process_0$ initiates I/O
4	Blocked	Running	$Process_0$ is blocked,
5	Blocked	Running	so $Process_1$ runs
6	Blocked	Running	
7	Ready	Running	I/O done
8	Ready	Running	$Process_1$ now done
9	Running	–	
10	Running	–	$Process_0$ now done

Process Control Block (PCB)

- In the kernel, each process is associated with a **process control block**
  - process number (pid)
  - process state
  - **program counter (PC)**
  - CPU registers
  - CPU scheduling information
  - memory-management data
  - accounting data
  - I/O status
- Linux's PCB is defined in struct task_struct: http://lxr.linux.no/linux+v3.2.35/include/linux/sched.h#L1221

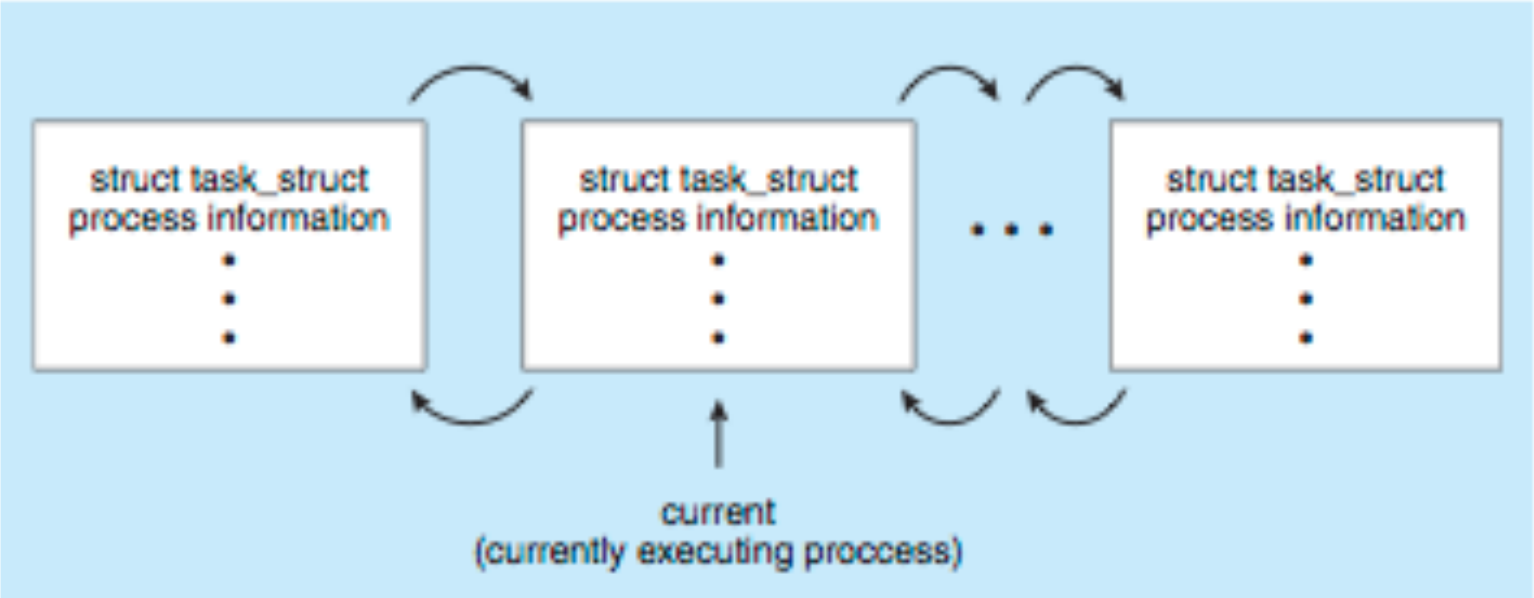
Process Control Block (PCB)



Process Control Block in Linux

- Represented by the C structure **task_struct**

```
pid_t pid;                       /* process identifier */
long state;                      /* state of the process */
unsigned int time_slice          /* scheduling information */
struct task struct *parent;      /* this process's parent */
struct list head children;       /* this process's children */
struct files struct *files;      /* list of open files */
struct mm_struct *mm;            /* address space of this process*/
...
```




```
current->state = new_state;
```

Threads

- So far, process has a single thread of execution
- Consider having multiple program counters per process
  - **Multiple locations** can execute at once
    - Multiple threads of control -> threads
- Must then have storage for thread details, multiple program counters in PCB

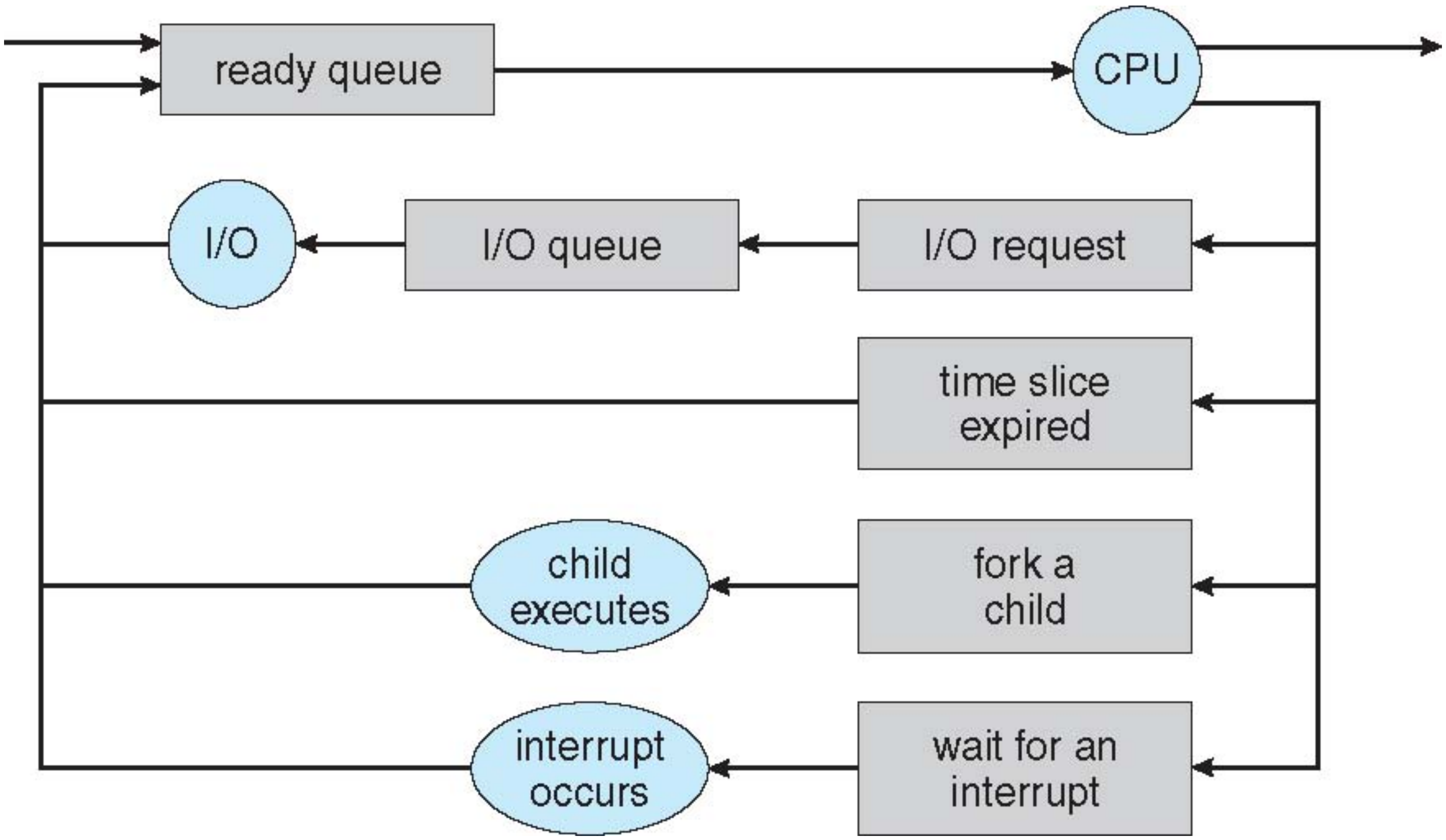
Process Scheduling

- **CPU scheduler** selects which process should be executed next and allocates CPU
    - invoked very frequently, usually in milliseconds: it must be fast

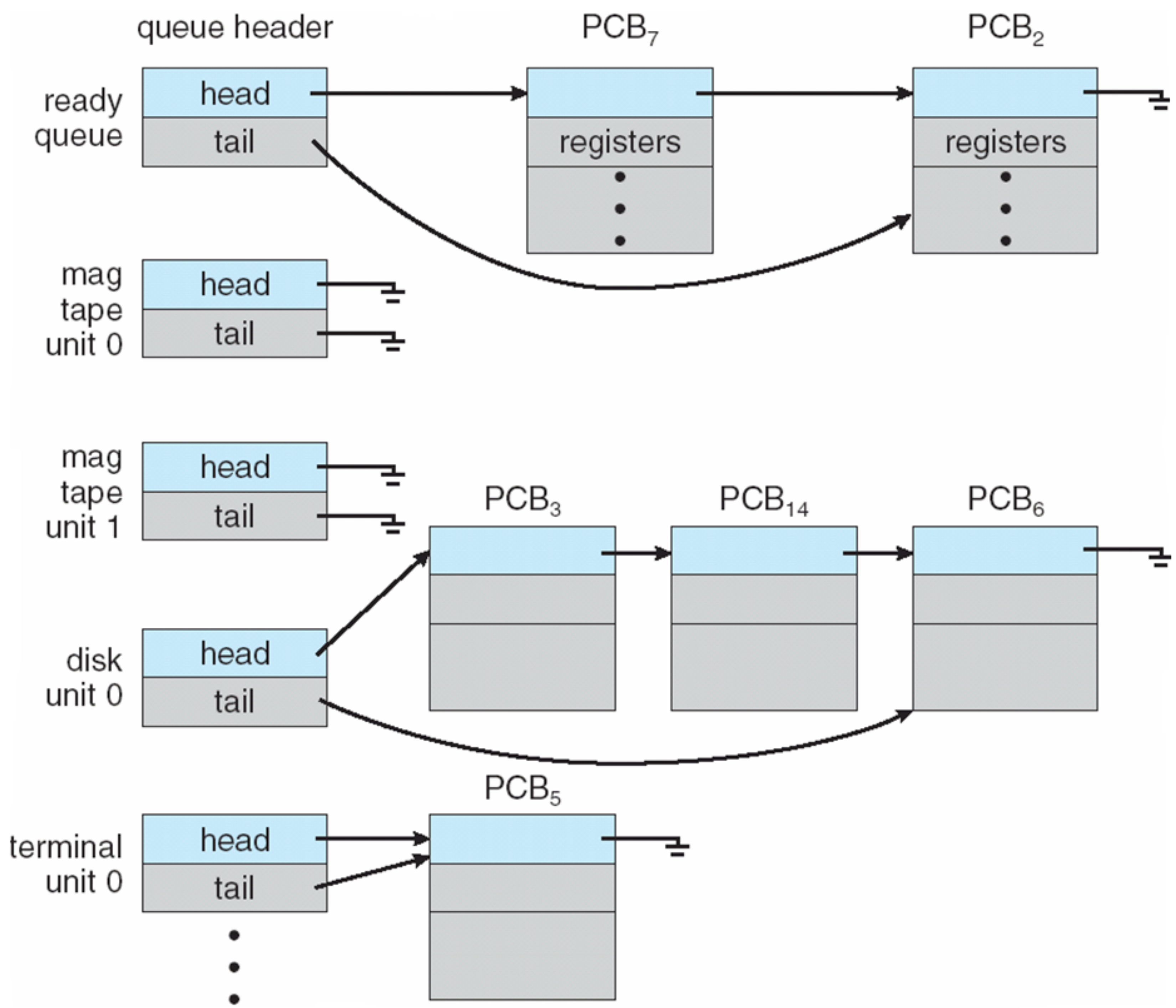
Process Scheduling

- To maximize CPU utilization, **kernel quickly switches processes onto CPU for time sharing**
- Process **scheduler** selects among available processes for next execution on CPU
- Kernel maintains scheduling queues of processes:
  - **job queue**: set of all processes in the system
  - **ready queue**: set of all processes residing in main memory, **ready and waiting to execute**
  - **device queues**: set of processes **waiting for an I/O device**
- **Processes migrate among the various queues**

Queues for Process Scheduling

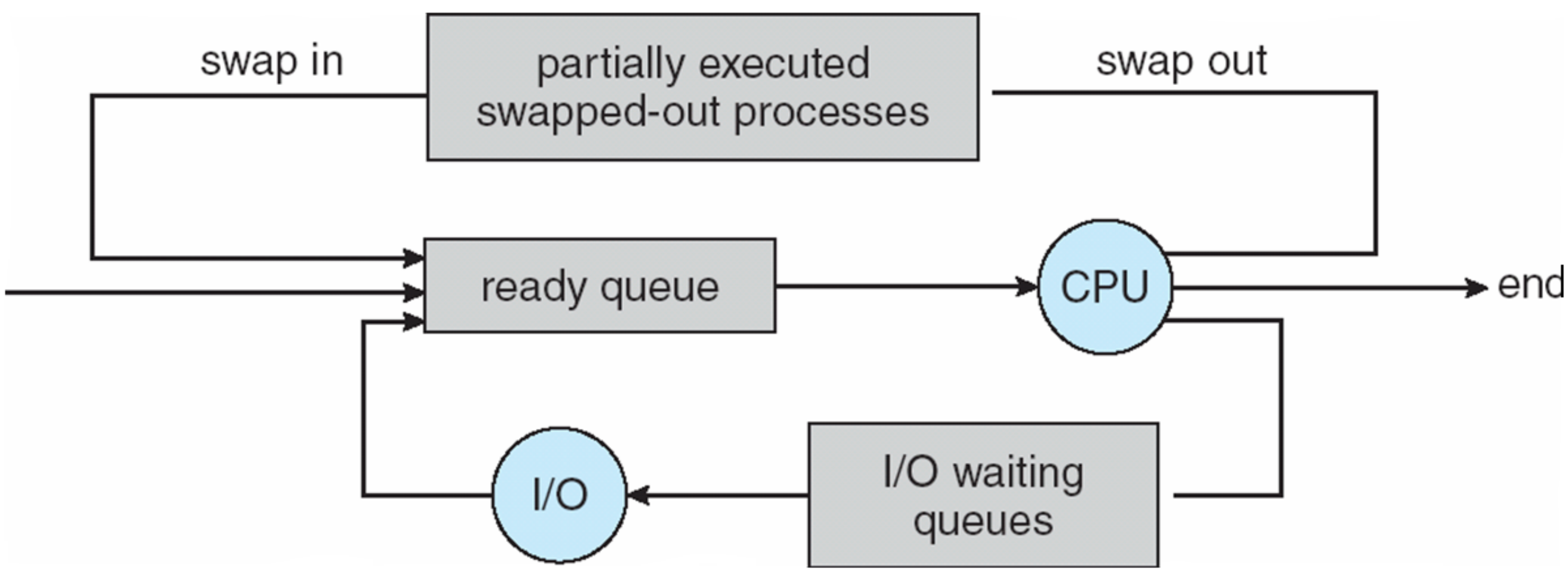


Ready Queue And Device Queues



Swap In/Out

- **Mid-term scheduler**
  - swap in/out partially executed process to relieve memory pressure



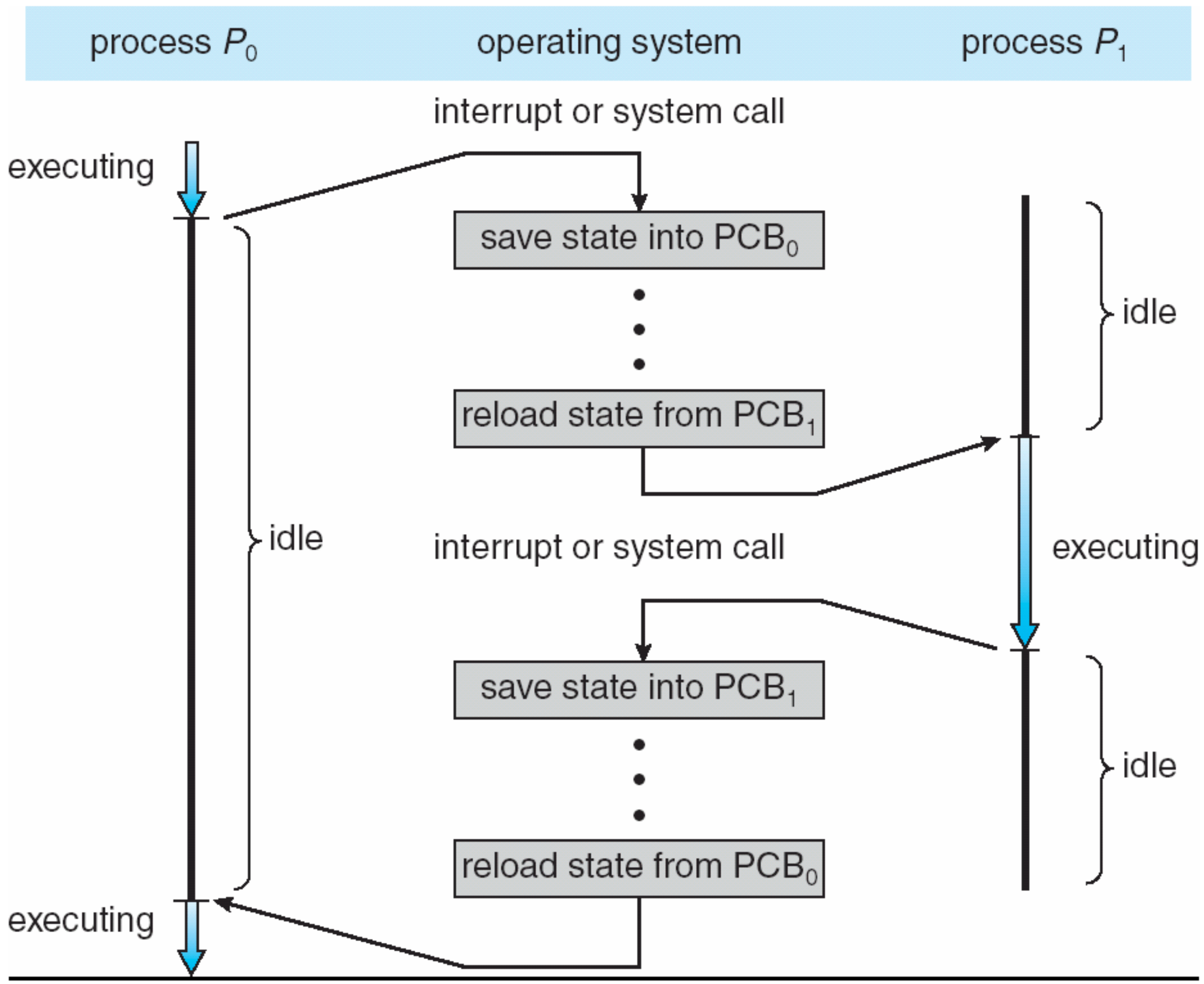
Scheduler

- Scheduler needs to balance the needs of:
  - **I/O-bound** process
    - spends more time doing I/O than computations
    - many short CPU bursts
  - **CPU-bound** process
    - spends more time doing computations
    - few very long CPU bursts

Context Switch

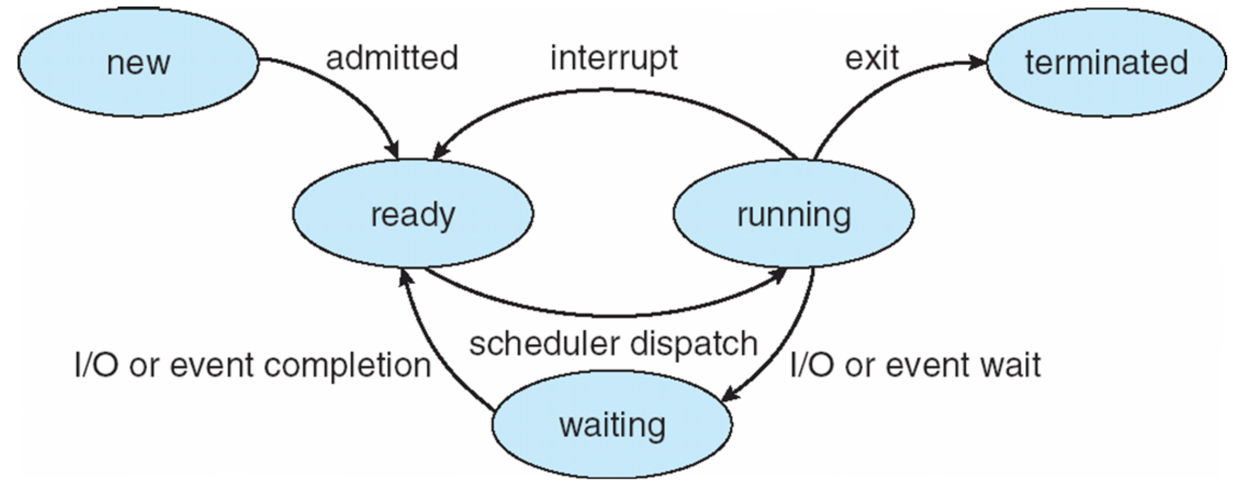
- **Context switch**: the kernel switches to another process for execution
    - save the state of the old process
    - load the saved state for the new process
- **Context-switch is overhead**; CPU does no useful work while switching
    - the more complex the OS and the PCB, longer the context switch
- Context-switch time depends on hardware support
    - some hardware provides multiple sets of registers per CPU: multiple contexts loaded at once

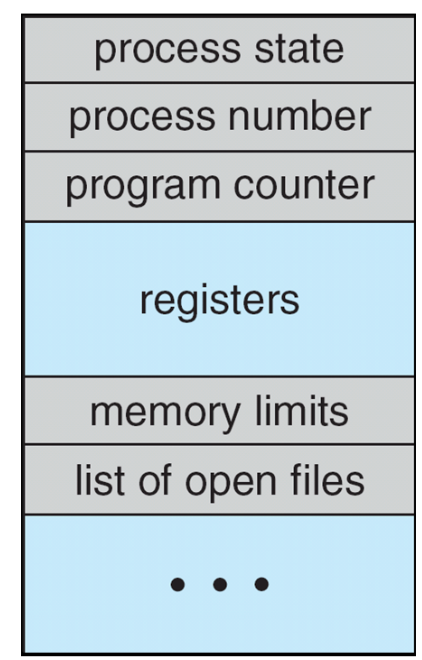
Context Switch



Review

- Process in memory
  - text, stack, heap, data
- Process state
  - new, ready, running, waiting, terminated
- Process control block (PCB)
- Context switch





Process Creation

- Parent process creates children processes, which, in turn create other processes, forming **a tree of processes**
  - process identified and managed via a process identifier (pid)
- Design choices:
  - three possible levels of **resource sharing**: all, subset, none
  - parent and children's **address spaces**
    - child duplicates parent address space (e.g., Linux)
    - child has a new program loaded into it (e.g., Windows)
  - **execution** of parent and children
    - parent and children execute concurrently
    - parent waits until children terminate

Process Creation

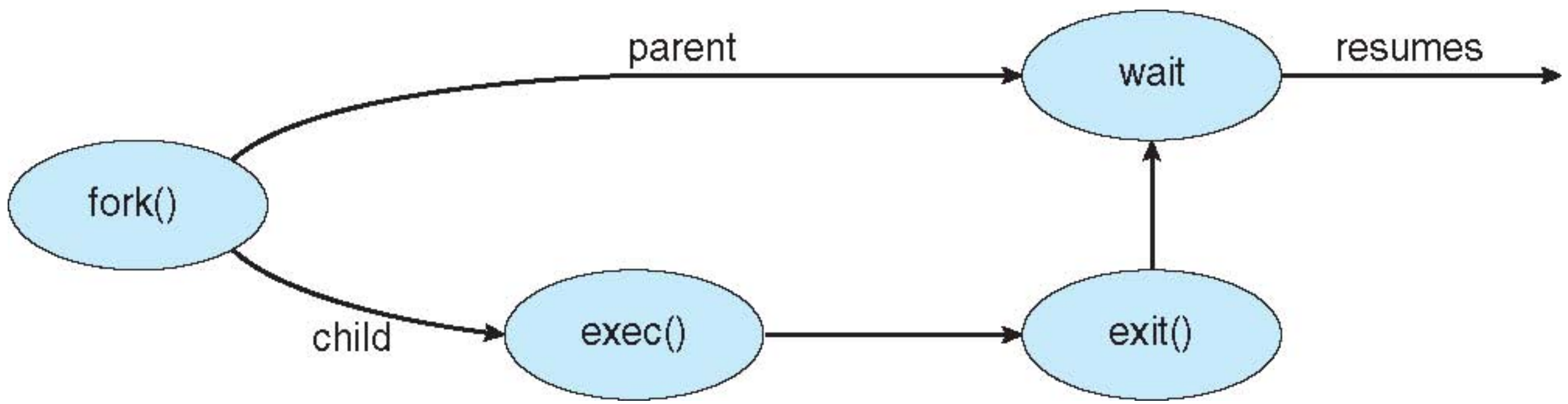
- UNIX/Linux system calls for process creation
  - **fork** creates a new process
  - **exec** overwrites the process' address space with a new program
  - **wait** waits for the child(ren) to terminate

What's the benefit of separating fork and exec?

C Program Forking Separate Process

```
#include <sys/types.h>
#include <studio.h>
#include <unistd.h>
int main()
{
   pid_t  pid;
   pid = fork();                        /* fork another process */
   if (pid < 0) {                       /* error occurred while forking */
      fprintf(stderr, "Fork Failed");
      return -1;
   } else if (pid == 0) {               /* child process */
      execlp("/bin/ls", "ls", NULL);
   } else {                             /* parent process */
      wait (NULL);
      printf ("Child Complete");
   }
   return 0;
}
```

Process Creation



Process Termination

- Process executes last statement and asks the kernel to delete it (**exit**)
  - OS delivers the return value from child to parent (via **wait**)
  - process' resources are deallocated by operating system
- Parent may terminate execution of children processes (**abort**), for example:
  - child has exceeded allocated resources
  - task assigned to child is no longer required
  - if parent is exiting, some OS does not allow child to continue
    - all children (the sub-tree) will be terminated - **cascading termination**

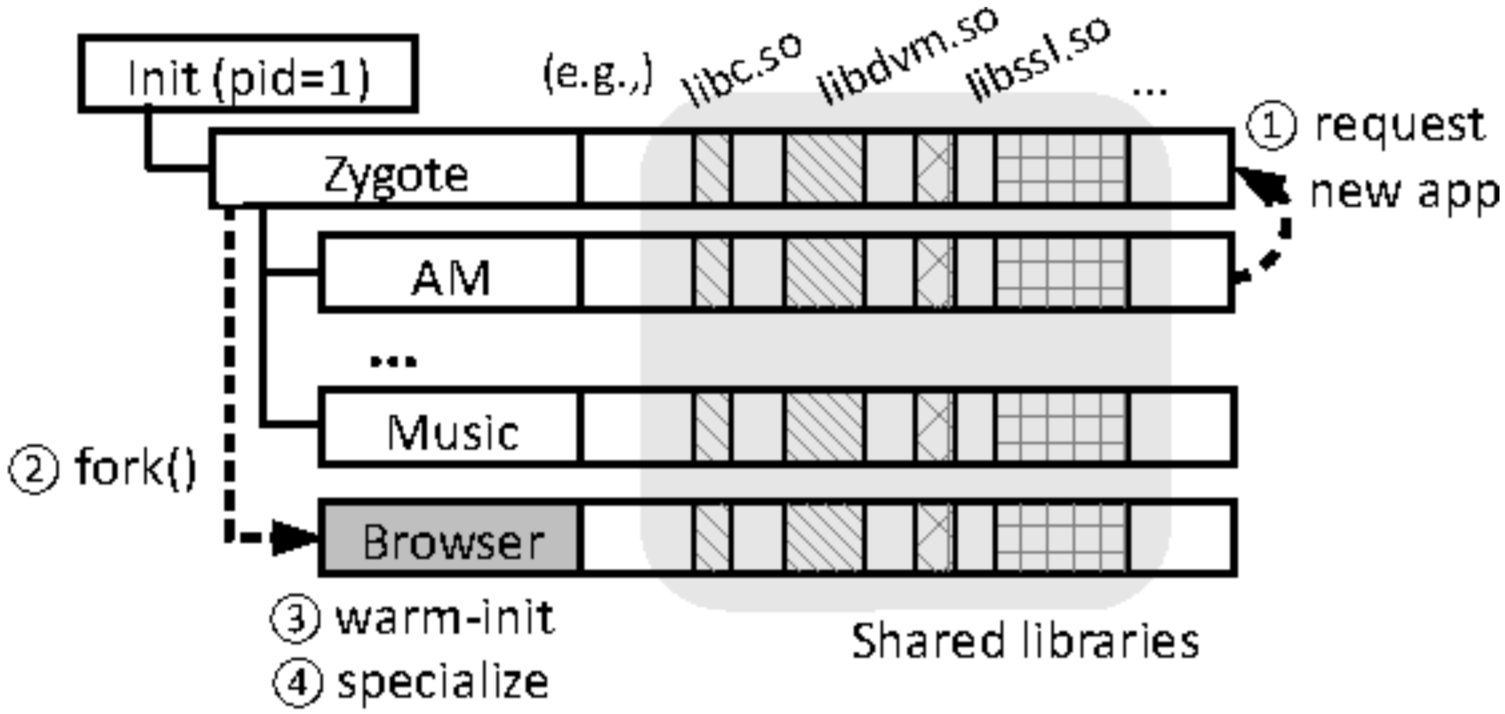
Zombie vs Orphan

- zombie vs orphan
  - When child process terminates, it is still in the process table until the parent process calls wait()
    - zombie: child has terminated execution, but parent did not invoke wait()
    - orphan: parent terminated without invoking wait - Systemd will take over. Systemd will call wait() periodically

Android Process Importance Hierarchy

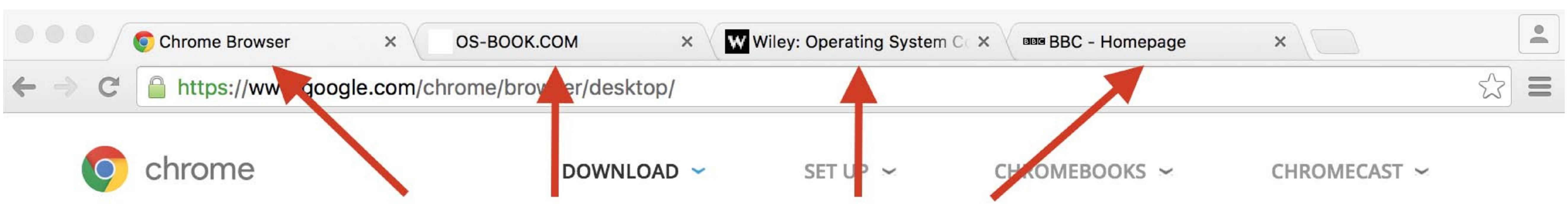
- Mobile operating systems often have to terminate processes to reclaim system resources such as memory. From most to least important:
  - Foreground process: visible on screen
  - Visible process: not directly visible, but performing activity that foreground process is referring
  - Service process: streaming music
  - Background process: performing activity, not apparent to the user
  - Empty process: hold no activity
- Android will begin terminating processes that are least important

Android Zygote



Multiprocess Architecture – Chrome Browser

- Many web browsers ran as single process (some still do)
    - If one web site causes trouble, entire browser can hang or crash
- Google Chrome Browser is multiprocess with 3 different types of processes:
    - **Browser** process manages user interface, 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



Each tab represents a separate process.

Chrome on Android: Isolated Process

```
{% for i in range(num_sandboxed_services) %}
<service android:name="org.chromium.content.app.SandboxedProcessService{{ i }}"
    android:process=":sandboxed_process{{ i }}"
    android:permission="{{ manifest_package }}.permission.CHILD_SERVICE"
    android:isolatedProcess="true"
    android:exported="{{sandboxed_service_exported|default(false)}}"
    {% if (sandboxed_service_exported|default(false)) == 'true' %}
    tools:ignore="ExportedService"
    {% endif %}
    {{sandboxed_service_extra_flags|default('')}} />
{% endfor %}
```

source: Remotely Compromising iOS via Wi-Fi and Escaping the Sandbox

Chrome on Android: Isolated Process

- Isolated process was introduced around Android 4.3
- "*If set to true, this service will run under a special process that is isolated from the rest of the system and has no permissions of its own.*"
- Chromium render process

```
$ adb shell ps -Z | grep chrome                                          [22:53:22]
u:r:untrusted_app:s0:c512,c768 u0_a39    7215  520   com.android.chrome
u:r:isolated_app:s0:c512,c768  u0_i0     7243  520   com.android.chrome:sandboxe
d_process0
u:r:untrusted_app:s0:c512,c768 u0_a39    7272  520   com.android.chrome:privileg
ed_process0
```

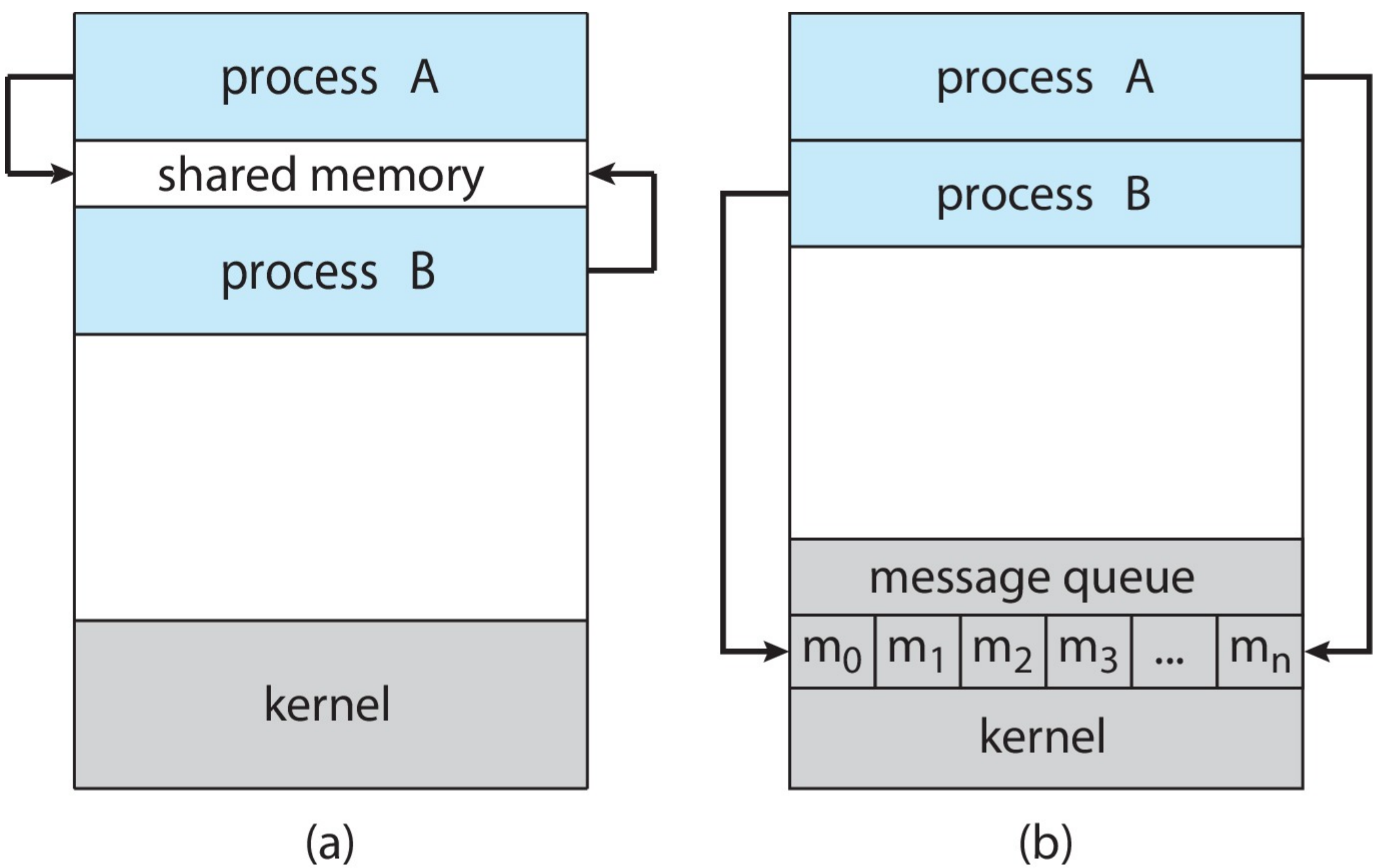
source: Remotely Compromising iOS via Wi-Fi and Escaping the Sandbox

Interprocess Communication

- Processes within a system may be independent or cooperating
  - **independent process**: process that cannot affect or be affected by the execution of another process
  - **cooperating process**: processes that can affect or be affected by other processes, including sharing data
    - reasons for cooperating processes: information sharing, computation speedup, modularity, convenience, Security
- Cooperating processes need **interprocess communication** (IPC)
- Two models of IPC
  - Shared memory
  - Message passing

Communications Models

(a) Shared memory. (b) Message passing.



Cooperating Processes

- **Independent process** cannot affect or be affected by the execution of another process
- **Cooperating process** can affect or be affected by the execution of another process
  - Advantages of process cooperation
  - Information sharing
  - Computation speed-up
  - Modularity
  - Convenience

Producer-Consumer Problem

- Paradigm for cooperating processes, producer process produces information that is consumed by a consumer process
    - **unbounded-buffer** places no practical limit on the size of the buffer
    - **bounded-buffer** assumes that there is a fixed buffer size

Interprocess Communication – Shared Memory

- An area of memory shared among the processes that wish to communicate
- The communication is under the control of the users processes not the operating system.
- Major issues is to provide mechanism that will allow the user processes to synchronize their actions when they access shared memory.
- Synchronization is an issue

Bounded-Buffer – Shared-Memory Solution

- Shared data

```
#define BUFFER_SIZE 10
typedef struct {
    . . .
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

Producer

```
item nextProduced;
while (true) {
    /* produce an item in nextProduced*/
    while (((in + 1) % BUFFER_SIZE) == out)
        ;   /* do nothing -- no free buffers */

    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER SIZE;
}
```

Consumer

```
item nextConsumed;
while (true) {
     while (in == out)
        ; // do nothing -- nothing to consume
     nextConsumed = buffer[out];
    out = (out + 1) % BUFFER SIZE;
    /*consume item in nextConsumed*/
}
```

- Solution is correct, but can only use **BUFFER_SIZE-1** elements
  - one unusable buffer to distinguish buffer full/empty

Message Passing

- Processes communicate with each other by exchanging messages
    - without resorting to shared variables
- Message passing provides two operations:
    - **send** (message)
    - **receive** (message)
- If P and Q wish to communicate, they need to:
    - **establish a communication link between them**
        - e.g., a mailbox(indirect) or pid-based(direct)
    - **exchange messages via send/receive**

Message Passing

- Direct communication
  - symmetry addressing: send(P, Message), receive(Q, Message)
  - asymmetry addressing: send(P, message), receive(id, Message)
- Indirect communication
  - send(A, Message), receive(A, Message)   mailbox A
- Mailbox can be implemented in both process and OS
  - Mailbox owner: who can receive the message

Message Passing: Synchronization

- Message passing may be either **blocking** or **non-blocking**
- Blocking is considered **synchronous**
  - **blocking send** has the sender block until the message is received
  - **blocking receive** has the receiver block until a message is available
- Non-blocking is considered **asynchronous**
  - **non-blocking send** has the sender send the message and continue
  - **non-blocking receive** has the receiver receive a valid message or null

Message Passing: Buffering

- **Queue of messages attached to the link**
  - **zero capacity:** 0 messages
    - sender must wait for receiver (rendezvous)
  - **bounded capacity**: finite length of n messages
    - sender must wait if link full
  - **unbounded capacity**: infinite length
    - sender never waits

POSIX Shared Memory

- POSIX Shared Memory
  - Process first creates shared memory segment

    shm_fd = **shm_open**(name, O CREAT | O RDWR, 0666);
  - Also used to open an existing segment
  - Set the size of the object: **ftruncate**(shm_fd, 4096);
  - Use **mmap**() to memory-map a file pointer to the shared memory object
  - Reading and writing to shared memory is done by using the pointer returned by mmap().

IPC POSIX Producer

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* strings written to shared memory */
const char *message_0 = "Hello";
const char *message_1 = "World!";

/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* create the shared memory object */
   shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);

   /* configure the size of the shared memory object */
   ftruncate(shm_fd, SIZE);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);

   /* write to the shared memory object */
   sprintf(ptr,"%s",message_0);
   ptr += strlen(message_0);
   sprintf(ptr,"%s",message_1);
   ptr += strlen(message_1);

   return 0;
}
```

IPC POSIX Consum

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>

int main()
{
/* the size (in bytes) of shared memory object */
const int SIZE = 4096;
/* name of the shared memory object */
const char *name = "OS";
/* shared memory file descriptor */
int shm_fd;
/* pointer to shared memory obect */
void *ptr;

   /* open the shared memory object */
   shm_fd = shm_open(name, O_RDONLY, 0666);

   /* memory map the shared memory object */
   ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0);

   /* read from the shared memory object */
   printf("%s",(char *)ptr);

   /* remove the shared memory object */
   shm_unlink(name);

   return 0;
}
```

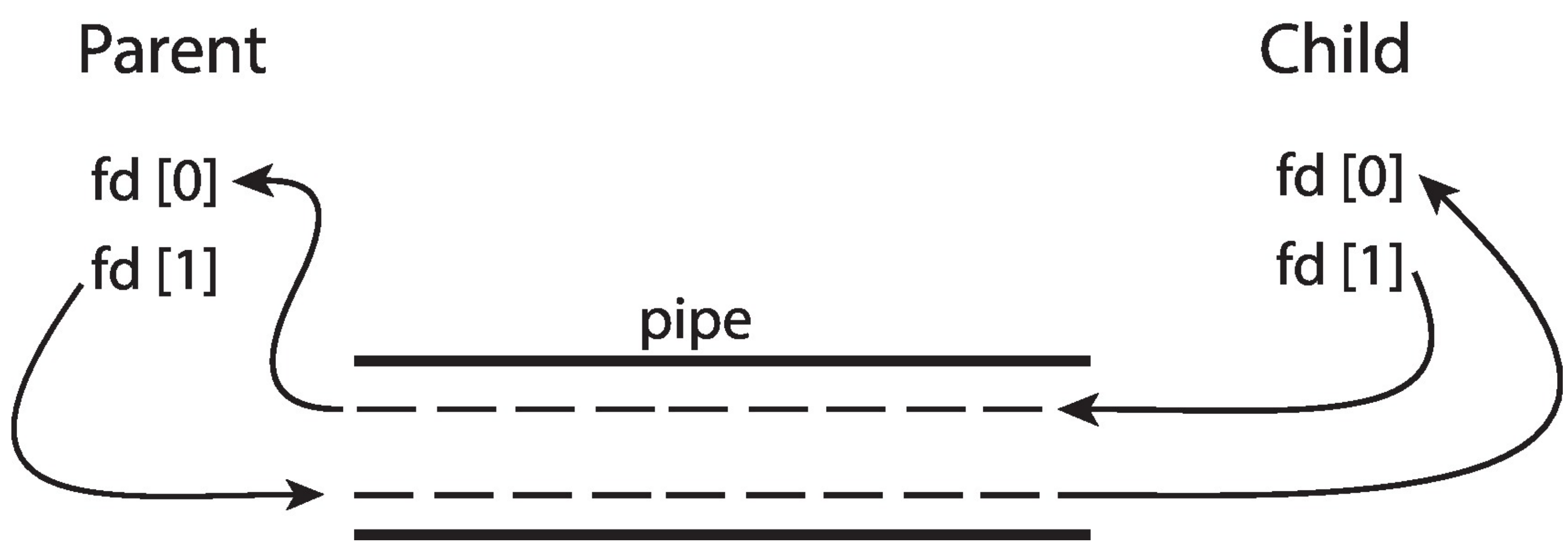
Pipes

- **Pipe** acts as a conduit allowing two local processes to communicate
- Issues
  - is communication unidirectional or bidirectional?
  - in the case of two-way communication, is it half or full-duplex?
  - must there exist a relationship (i.e. parent-child) between the processes?
  - can the pipes be used over a network?
    - usually only for local processes

Ordinary Pipes

- Ordinary pipes allow communication in the **producer-consumer** style
  - producer writes to one end (the write-end of the pipe)
  - consumer reads from the other end (the read-end of the pipe)
  - ordinary pipes are therefore **unidirectional**
  - Two pipes are needed if we need bidirectional communication
- Require **parent-child relationship** between communicating processes
- Activity: review Linux **man pipe**

Ordinary Pipes



Named Pipes

- Named pipes are more powerful than ordinary pipes
  - communication is bidirectional
  - no parent-child relationship is necessary between the processes
  - several processes can use the named pipe for communication
- **Named pipe** is provided on both UNIX and Windows systems
  - On Linux, it is called FIFO

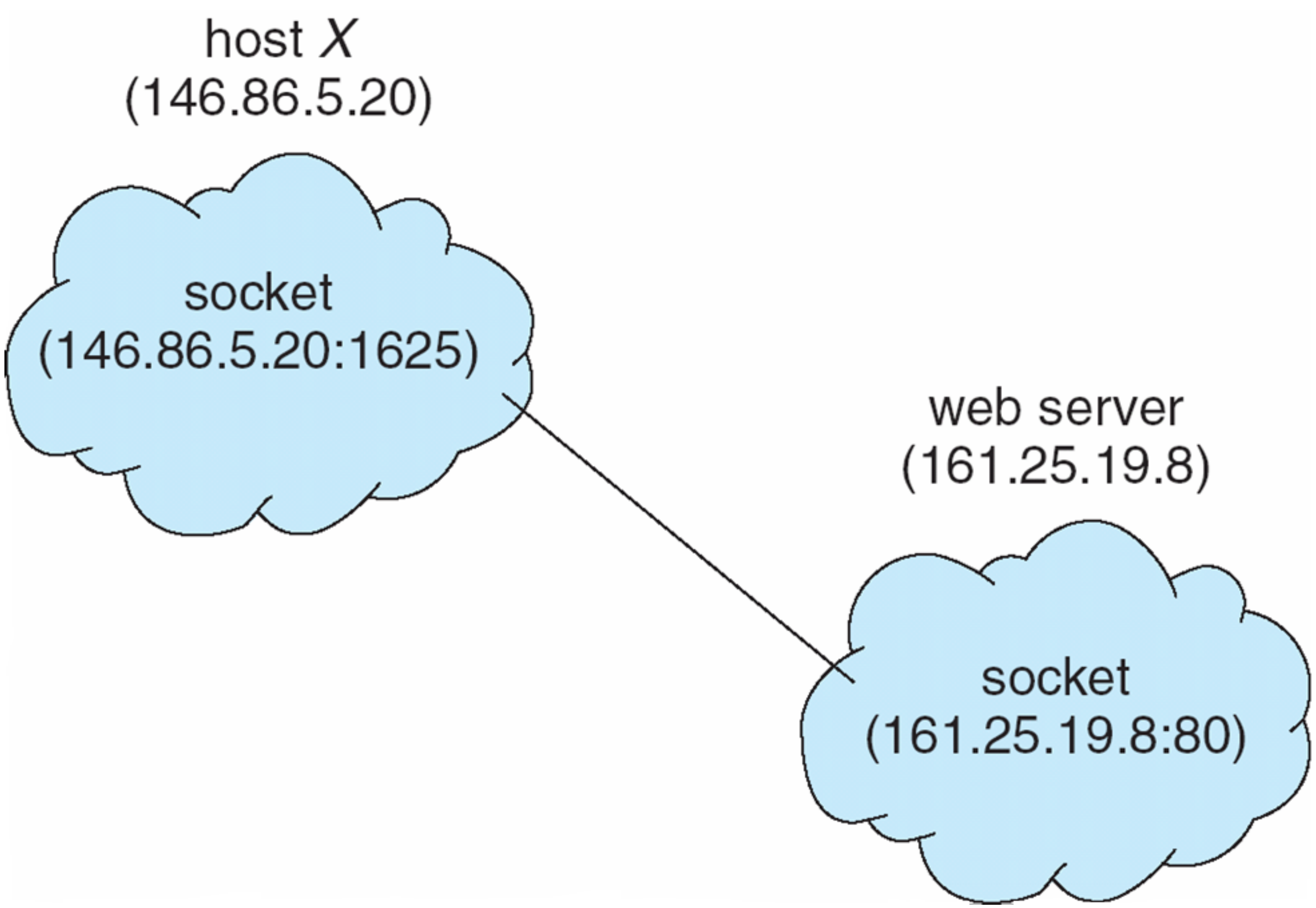
Client-server Communication

- Sockets
- Remote procedure calls
- Remote method invocation (Java)

Sockets

- A **socket** is defined as an endpoint for communication
  - concatenation of IP address and port
  - socket 161.25.19.8:1625 refers to port 1625 on host 161.25.19.8
- Communication consists between **a pair of sockets**

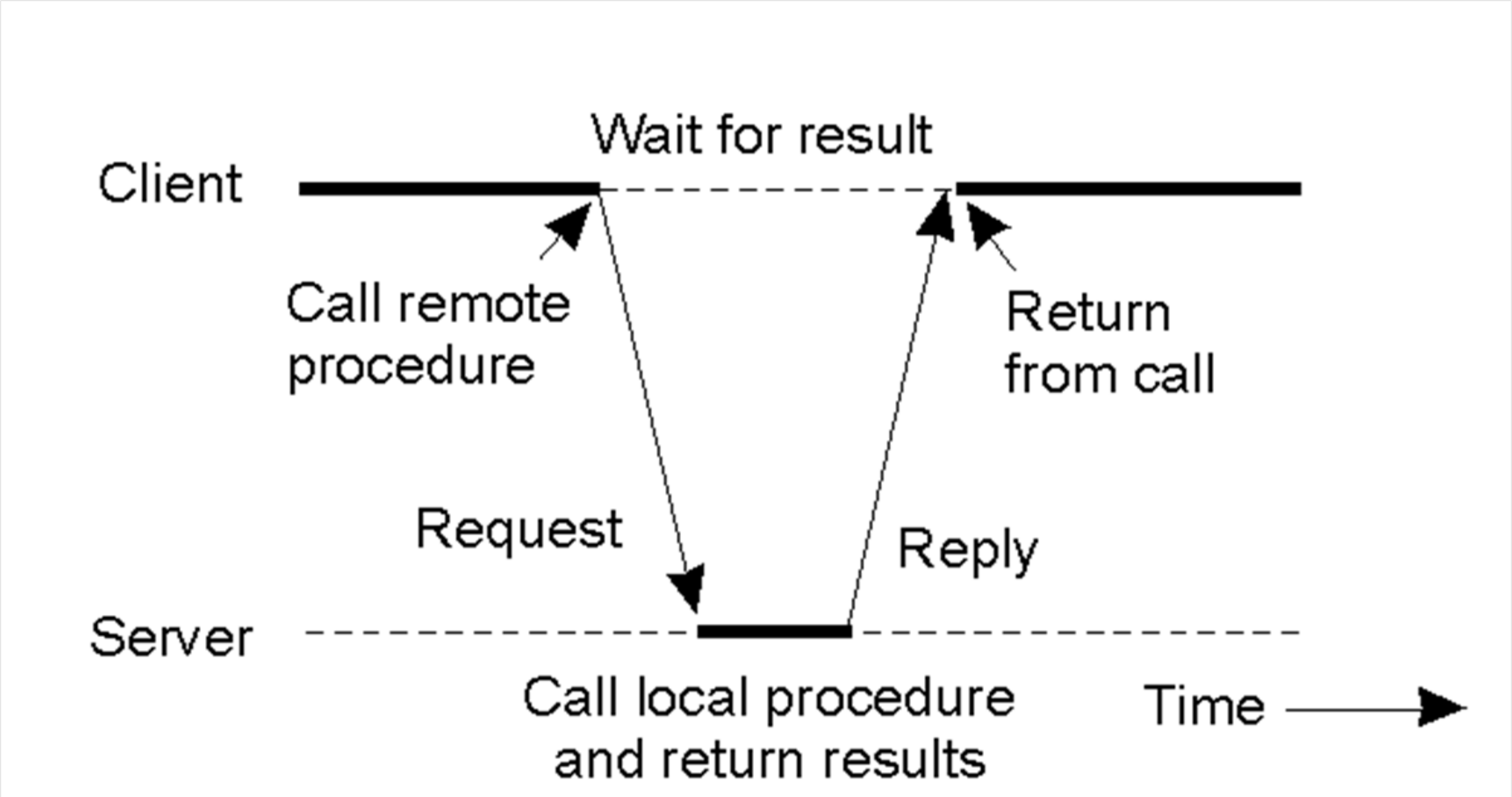
Socket Communication



Remote Procedure Call

- Remote procedure call (RPC) abstracts function calls between processes across networks (or **even local processes**)
- **Stub**: a proxy for the actual procedure on the **remote machine**
  - client-side stub locates the server and **marshalls** the parameters
  - server-side stub receives this message, **unpacks** the marshalled parameters, and performs the procedure on the server
  - return values are marshalled and sent to the client

Remote Procedure Call



A Simple Kernel Module

```c
#include <linux/init.h>
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/printk.h>
#include <linux/sched.h>
#include <linux/sched/signal.h>

MODULE_LICENSE("GPL");
MODULE_AUTHOR("Yajin Zhou");
MODULE_DESCRIPTION("A simple example Linux module.");
MODULE_VERSION("0.01");


static int __init os_lkm_example_init(void) {
        struct task_struct *task;

        for_each_process(task)
                printk(KERN_INFO "%s [%d]\n", task->comm, task->pid);

    return 0;


}

static void __exit os_lkm_example_exit(void) {
        printk(KERN_INFO "Goodbye, World!\n");
}

module_init(os_lkm_example_init);
module_exit(os_lkm_example_exit);
```

Further reading: https://blog.sourcerer.io/writing-a-simple-linux-kernel-module-d9dc3762c234

HW3 is out!