CSC 139
Operating System Principles

Chapter 3
Process

Herbert G. Mayer, CSU CSC

8/1/2022

Syllabus

- Process Concept
- Process Scheduling
- Process States
- Operations on Processes
- Interprocess Communication (IPC)
- Examples of IPC Systems
- Client-Server Systems

Objectives

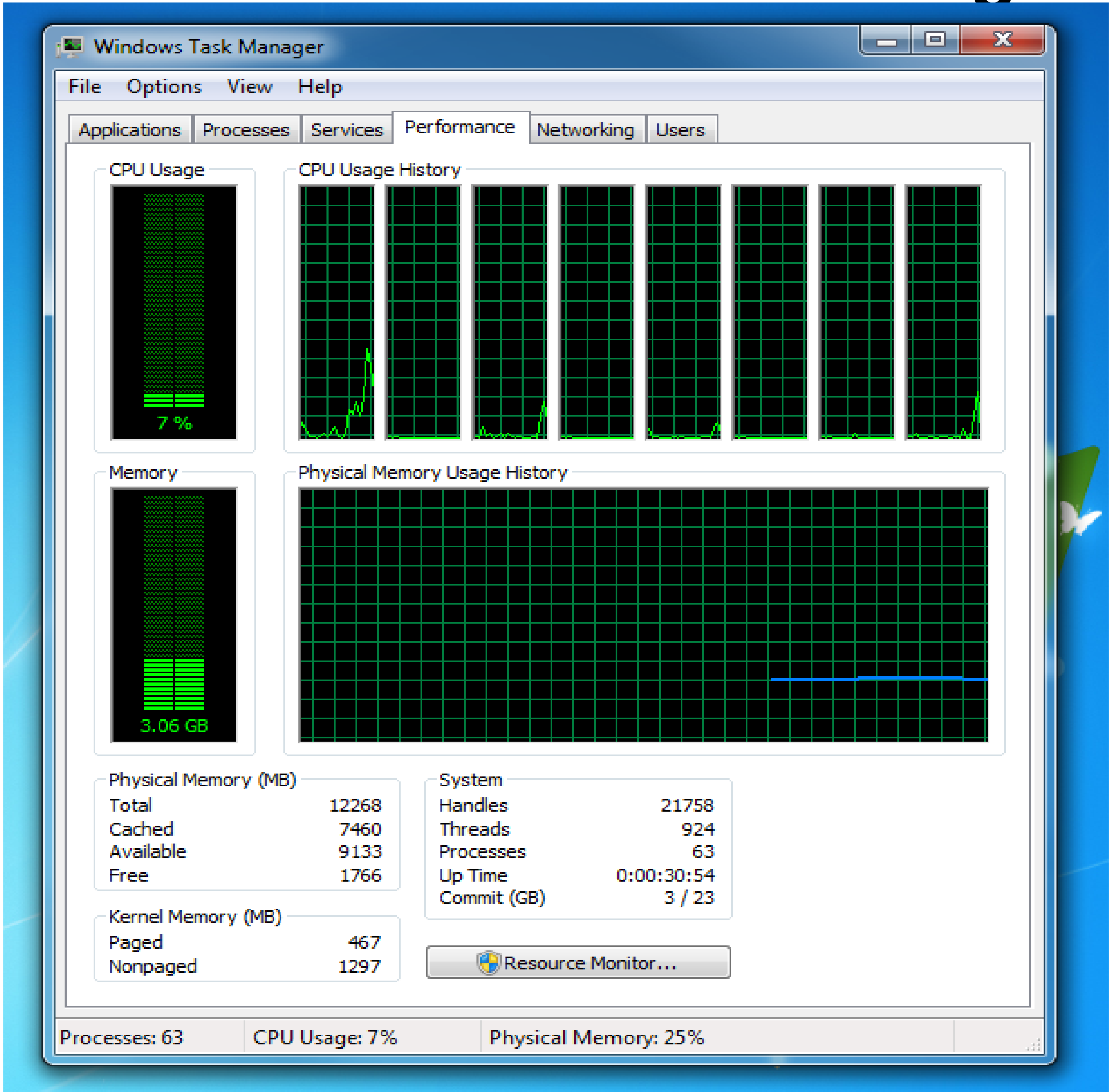
- Introduce process: Process is a program in execution, which forms the basis for computations, managed by an OS
- Describe various needs and features of processes: Scheduling, priorities, creation, execution, interruption, termination, and communication
- Explore interprocess communication using shared memory or message passing
- Sketch out communication in client-server systems
- Discuss industry- and academic standards that are process related!

Process Concept

Informally:

- Process is sometimes mistaken to be synonymous to program; but they do differ
- Program & process do share a key ingredient: object code
- Object code per-se is lifeless, just a collection of bits; some bits are data, others are executable code: but the key point is: executable, not yet executing! Not loaded!
- Process is a program that is executing, code that is currently running, has been loaded and is ready to run but is in the wait queue!
- To actually execute, a process needs resources e.g.: processor, loaded object code, memory to run in, may need files for reading data and generating data
- All managed under OS control!

Windows Process Manager



Process Concept

More Formally:

- **Process is a program in execution under OS control**
- **OS executes and manages processes in various modes:**
  - **Supervisor mode – to manage OS's resources, user processes**
  - **Batch mode – user jobs to be executed w/o user interaction**
  - **Time-shared mode – user interacts with running software**
- **Various process components:**
  1. **Program code, AKA text section** (in antique ☺ IBM terms)
  2. **HW resources, including registers, static memory, stack, heap**
  3. **Stack containing temporary & local data, varies in size during run, grows at call, shrinks at return, starts and ends at ~0 size**
  4. **Functions & function parameters, return address, local objects**
  5. **Data section containing global objects; possibly shared**
  6. **Files for data to be read (input) and generated (output)**

Process Concept

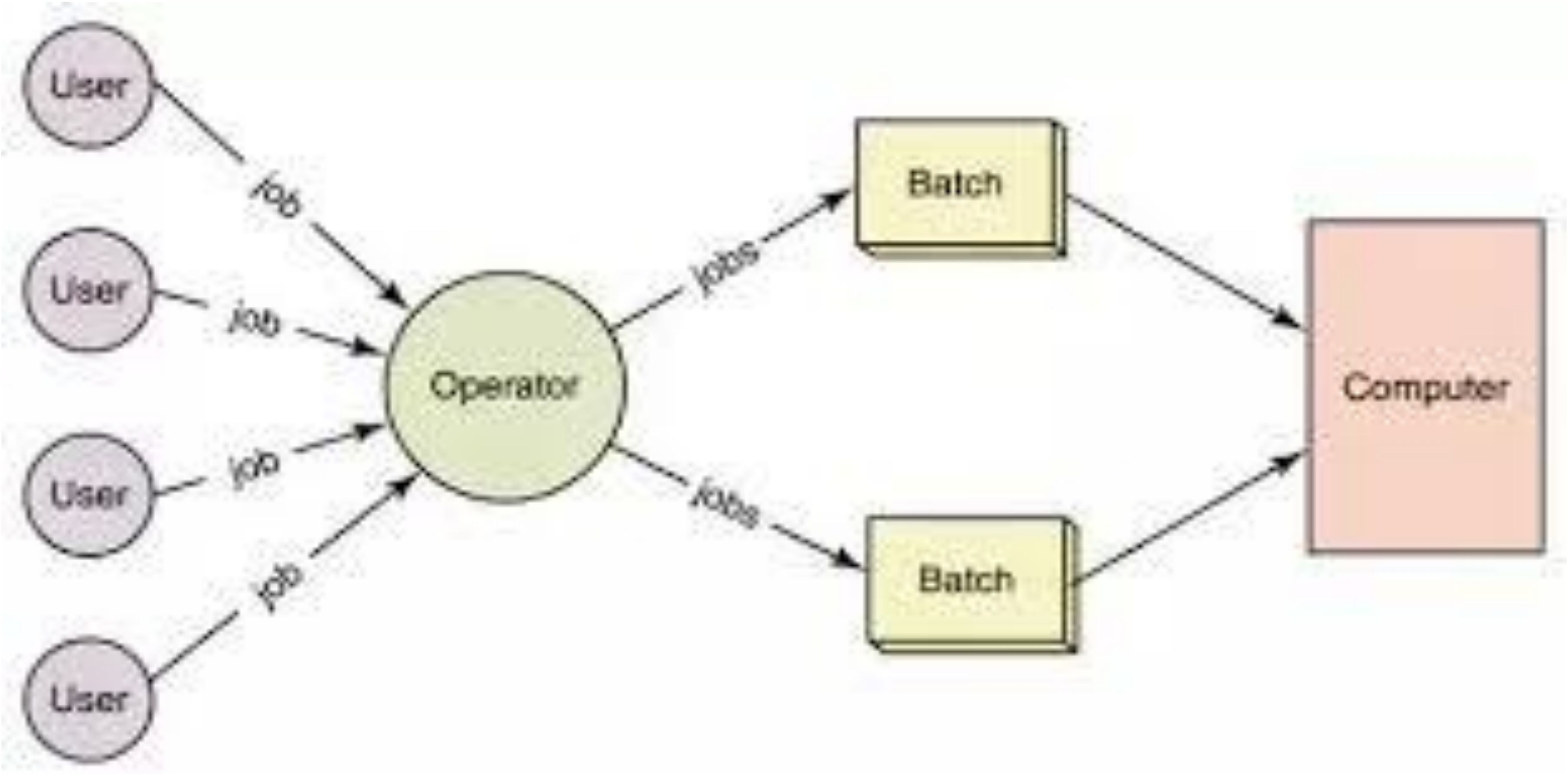
Remember ☺

Process is a program in execution!

- A program *per-se* is not alive, it is just a sequence of silent bits, stored somewhere, perhaps on disk (or solely in a programmer's mind ☺)
- Once a program is loaded into memory, ready to run, or actually running on a system, then it is alive, an active process, consuming resources, inhabiting a position in some scheduling queue, lasting a certain time, consuming input, producing output
- Then the program has become a process
- A process is alive
- A program only has the potential to live in the future; even a process in wait state is *alive*

Process Concept

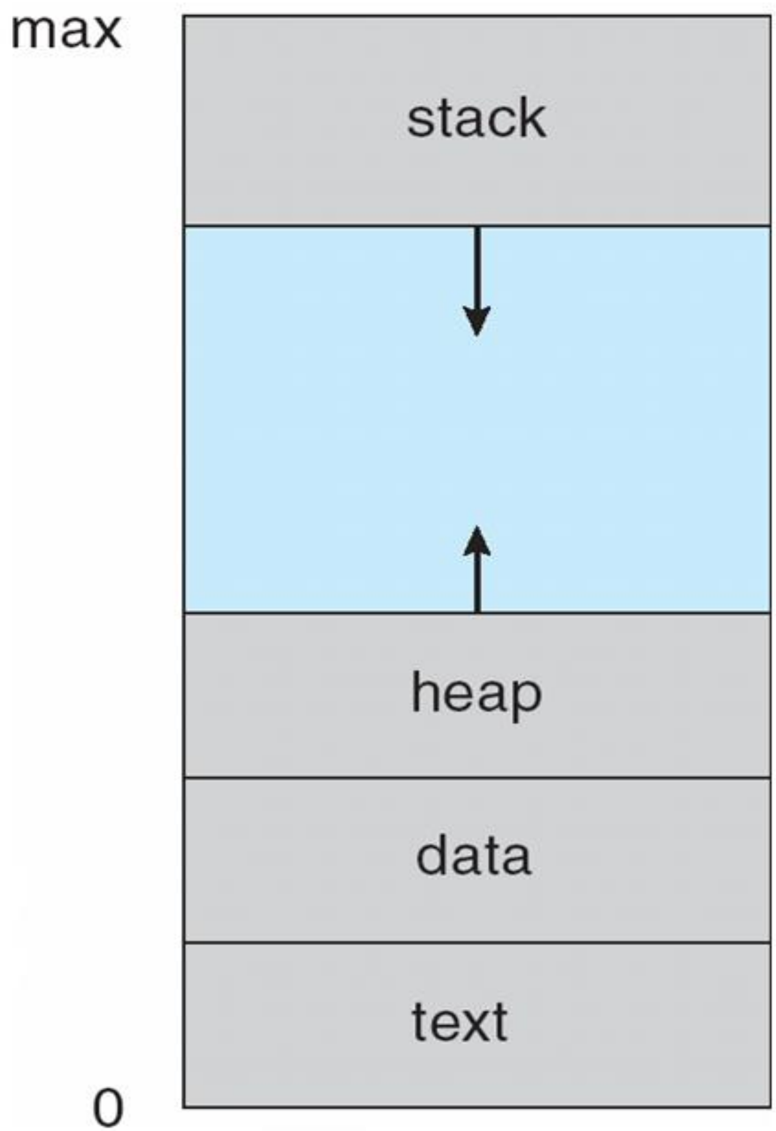
Batch Process, involves human operator; *in days of old*



Process vs. Program

- **Program is a *passive* entity, just bits (presumably meaningful) stored on a disk, as an executable file**
- **Process is *active, is alive, is an executing program***
  - **Program becomes process when executable file is loaded into memory, ready to be granted processor time**
- **Execution of program starts via 1. GUI mouse click, or 2. Command line entry of its name, or 3. Pushing a button to read cards, or 4. Touching an icon, etc.**
- **One program can evolve into several processes in various ways:**
  - **Consider multiple users executing the same program; such as an editor on a time-shared main frame: distinct processes**
  - **Or single user program, when executing, may fork into multiple sub-processes; desirable on MP target for speed**

Process in Memory

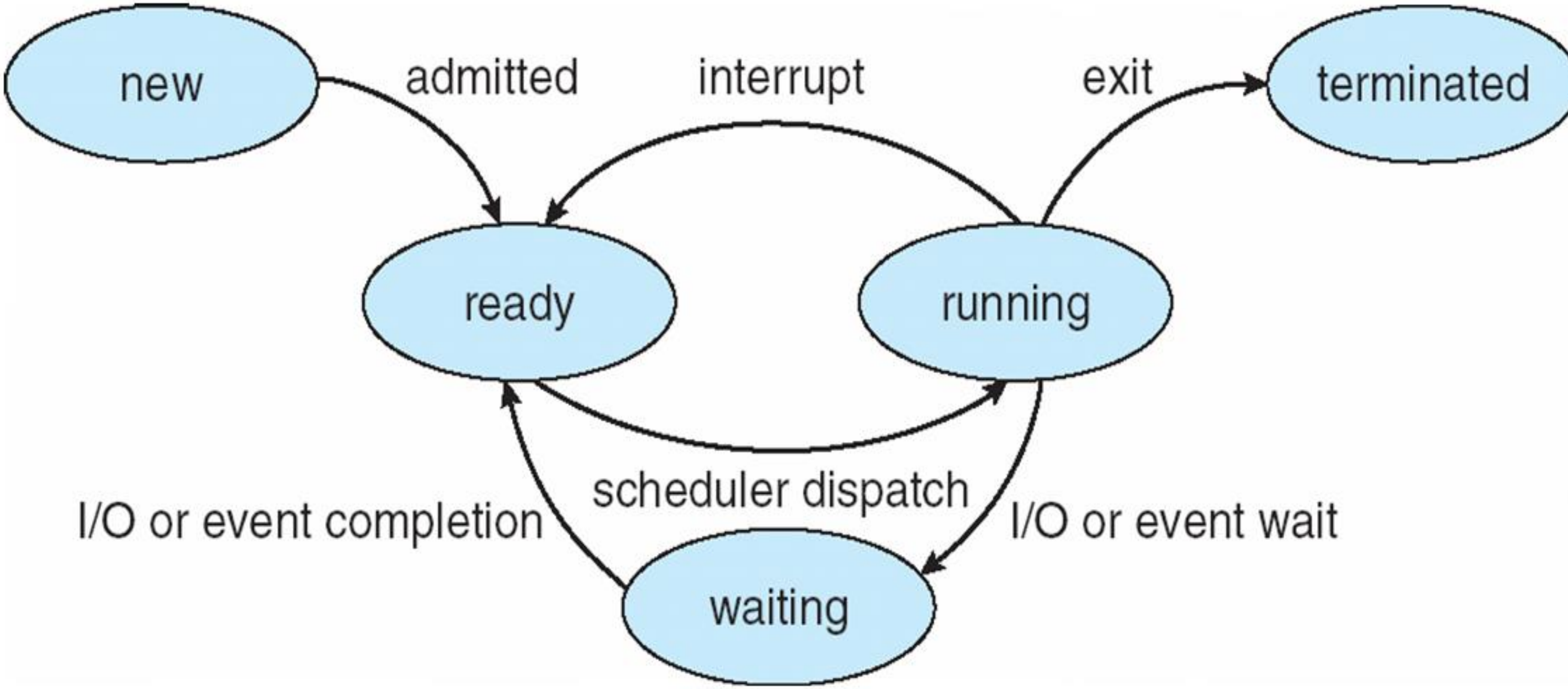


Process States

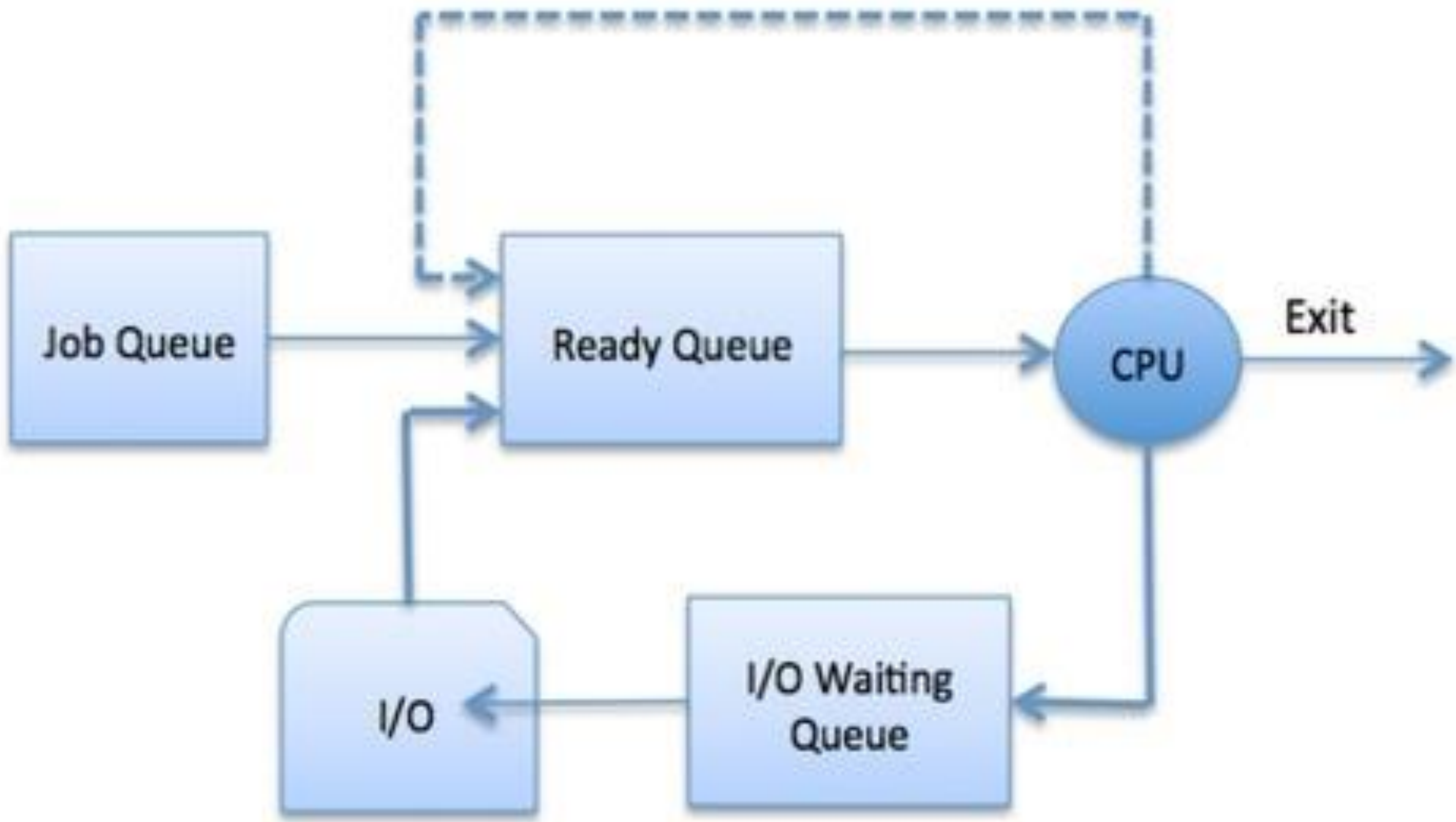
While a process executes, over time it changes state

- **new: Process is created by loading from disk into memory**
- **running: Instructions are being executed on a CPU**
- **waiting: Process is waiting for some event to occur on its behalf; could be data input, or another event; may change priority while waiting**
- **ready: Process is waiting to be assigned to a processor, i.e. it is ready run but not yet running, as it has not been granted a CPU**
- **terminated: Process has completed execution; its whole memory space is being freed; all other resources will be re-acquired by OS; PID will be purged**
- **A priori and a posteriori: the OS owns it all!**

Process States

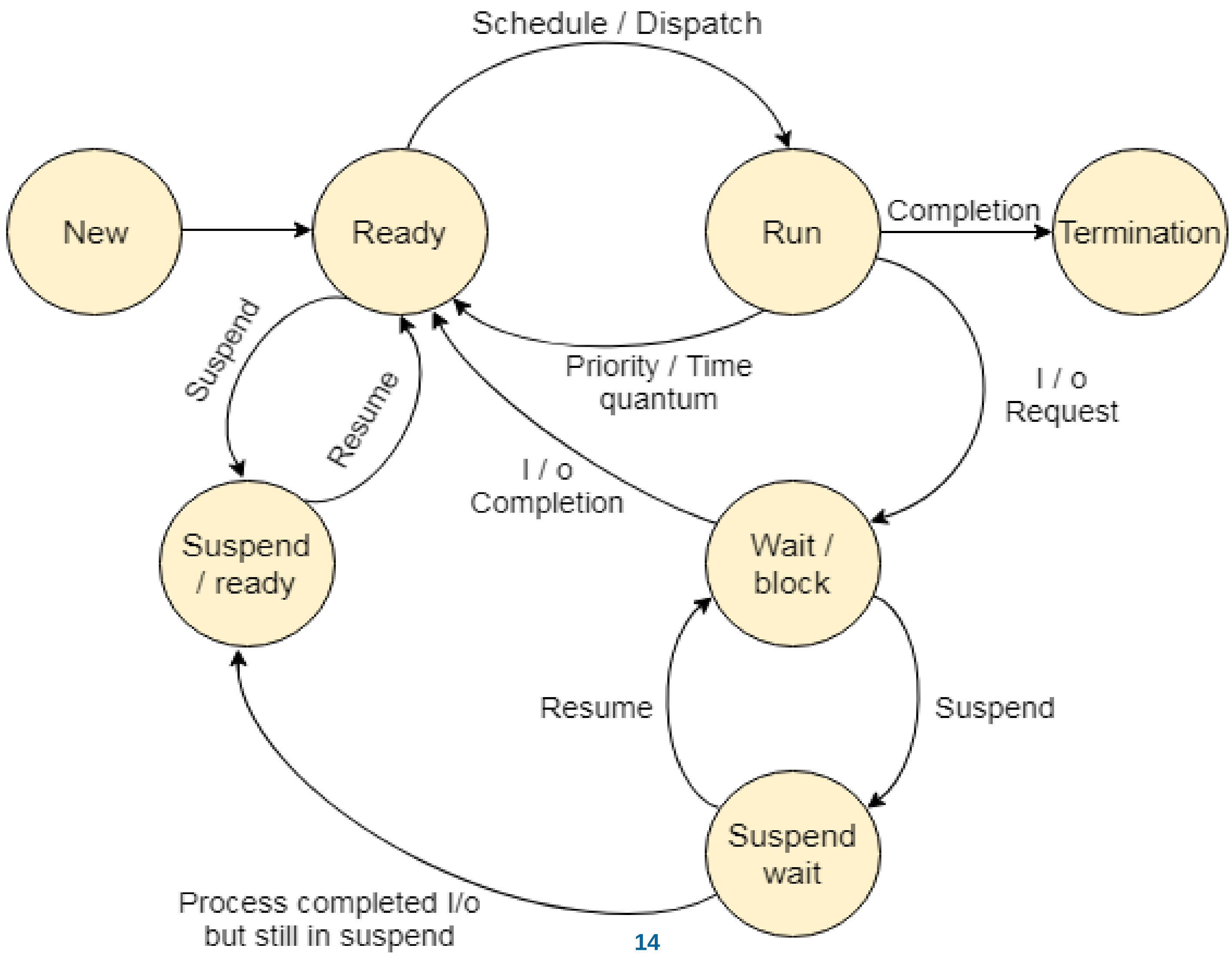


Process Queue



Process States

Same Process State Model, More Detail

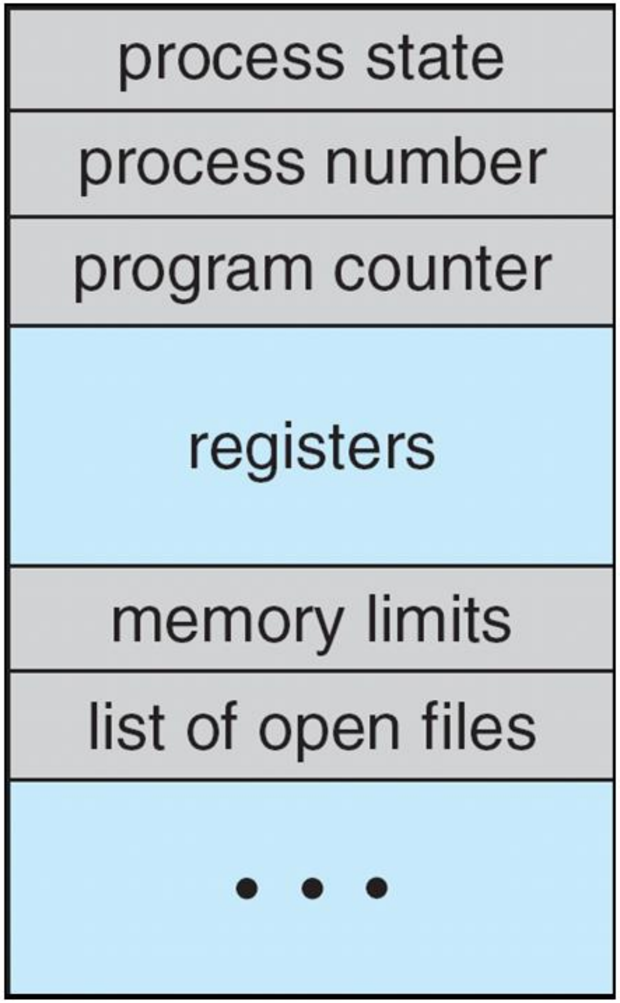


Process Control Block (PCB)

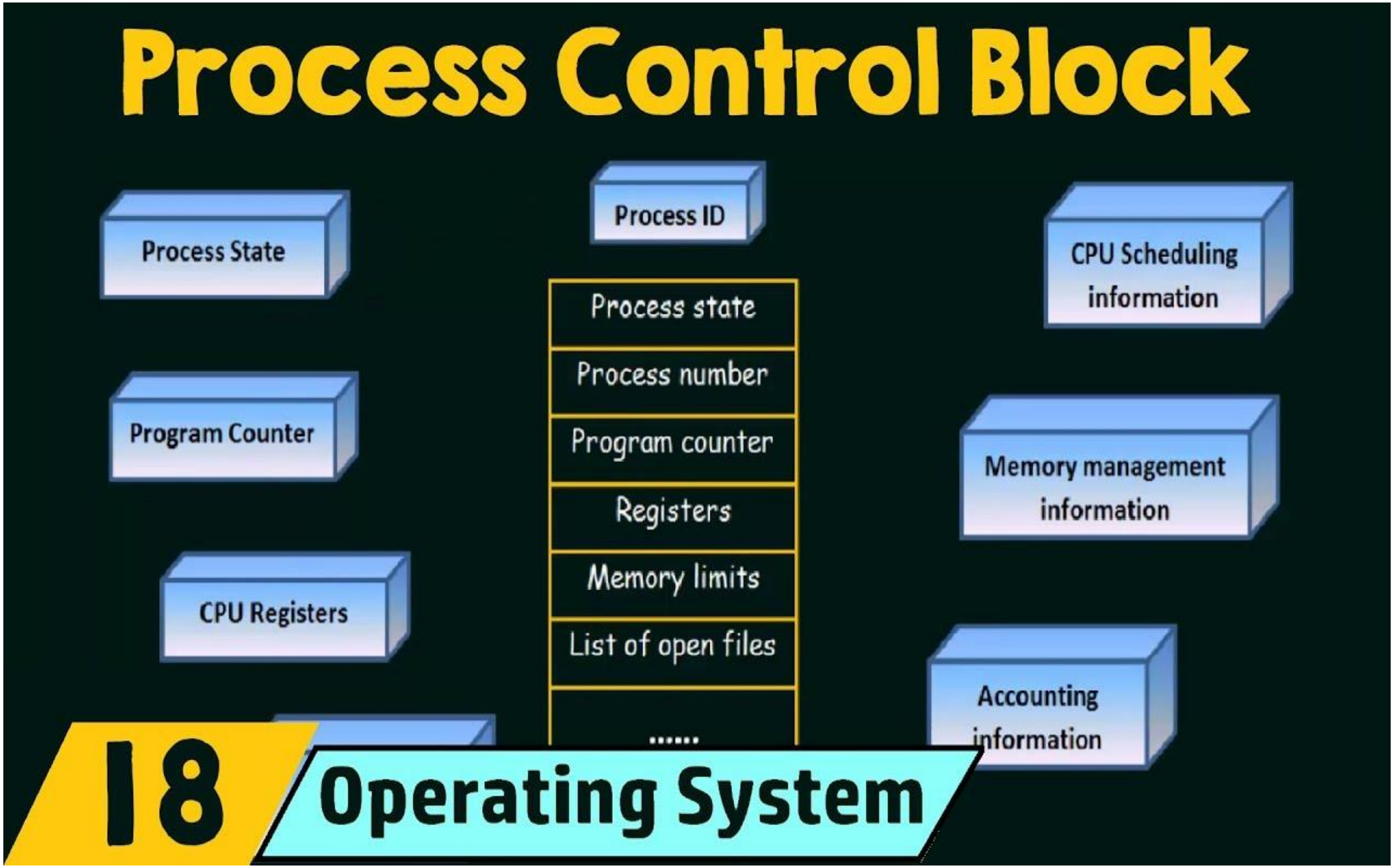
Information associated with process:
e.g. an actual C++ data structure:

AKA process control block PCB

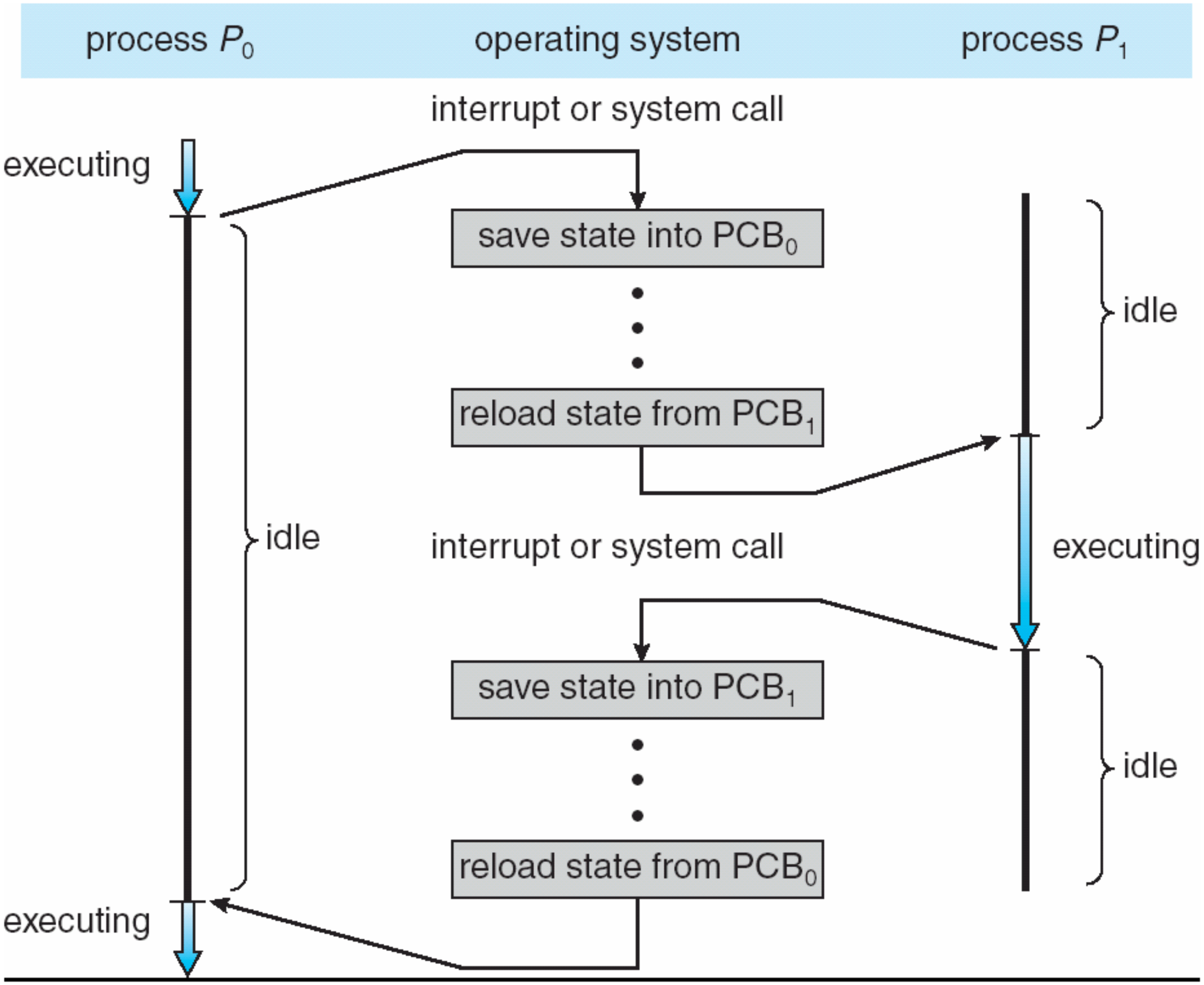
- **Process ID (AKA PID): unique number**
- **Process states: run, wait or suspend**
- **Program counter – address of next instruction to execute (AKA pc or ip)**
- **CPU registers – contents of all process-centric registers; generally: all regs**
- **CPU scheduling information – priority, scheduling queue pointers**
- **Memory management information – memory allocated to process**
- **Accounting info – CPU use, clock time elapsed since start; time limits if defined**
- **I/O status information – I/O devices allocated to process, list of open files**



PCB


Process Control Block
Process State
Process ID
CPU Scheduling information
Process state
Process number
Program counter
Registers
Memory limits
List of open files
......
Program Counter
Memory management information
CPU Registers
Accounting information
18
Operating System

CPU Switch *Process to Process*



CPU Switch *Process to Process*

- **Visible from above picture that inter-process switch causes process idle time: to load + restore PCB**
- **Idle time generally to be avoided, or minimized**
- **To reduce idle time caused by storing PCB info onto disk and then re-loading PCB from disk . . .**
- **Leave all PCB info in memory, even of suspended process, so that multiple disk accesses can be skipped during context switch**
- **Conflict: less free physical memory available for other active processes**
- **Typical for an OS: *conflicting goals***

CPU Switch *Process to Process*

- **Scheduler's switch from one process to another via interrupt is similar to function call + return**
- **Except during *process switch* the changing entities are unrelated, different processes being interrupted**
- **While in the function call + return mechanism all actions are part of the same program**
- **But after some "distraction"**
  - **Once the calling function experiences the callee's return . . .**
  - **Once the interrupted process receives a new time slice . . .**
- **Execution continues where it had left off before the change in execution flow: transparently!**

Threads & Scheduling

Threads

- **Process often viewed as a single thread of execution**
- **For Single Thread, the thread notion is not meaningful!**
- **Consider executing multiple threads, using multiple program counters (pc), one pc per thread, all progressing as part of one single program:**
  - ***Multiple code locations*** **can sometimes execute simultaneously on MP target; or concurrently on UP target; even that can speed up execution! But how?**
  - **By reducing “wait” conditions that otherwise would have happened, costing real idle time for process; not for CPU**
  - ***Multiple control sequences*** **can execute ➔ threads**
  - **And still just one single, original program is executing!**
- **Can any and every part of process become a thread?**

Threads

- **Process often viewed as a single thread of execution**
- **For Single Thread, the thread notion is not meaningful!**
- **Consider executing multiple threads, using multiple program counters (pc), one pc per thread, all progressing as part of one single program:**
  - ***Multiple code locations* can sometimes execute simultaneously on MP target; or concurrently on UP target; even that can speed up execution! But how?**
  - **By reducing “wait” conditions that otherwise would have happened, costing real idle time for process; not for CPU**
  - ***Multiple control sequences* can execute → threads**
  - **And still just one single, original program is executing!**
- **Can any and every part of process become a thread?**
- **Only logically independent sections can! Detail later…**

Linux Task

Virtual Memory data structure of Linux:

```
// mm_struct describes virtual memory of a task or process
struct mm_struct {
  int      count;
  pgd_t     * pgd;
  unsigned long context;
  unsigned long start_code, end_code, start_data, end_data;
  unsigned long start_brk, brk, start_stack, start_mmap;
  unsigned long arg_start, arg_end, env_start, env_end;
  unsigned long rss, total_vm, locked_vm;
  unsigned long def_flags;
  struct   vm_area_struct * mmap;
  struct   vm_area_struct * mmap_avl;
  struct   semaphore mmap_sem;
};
```

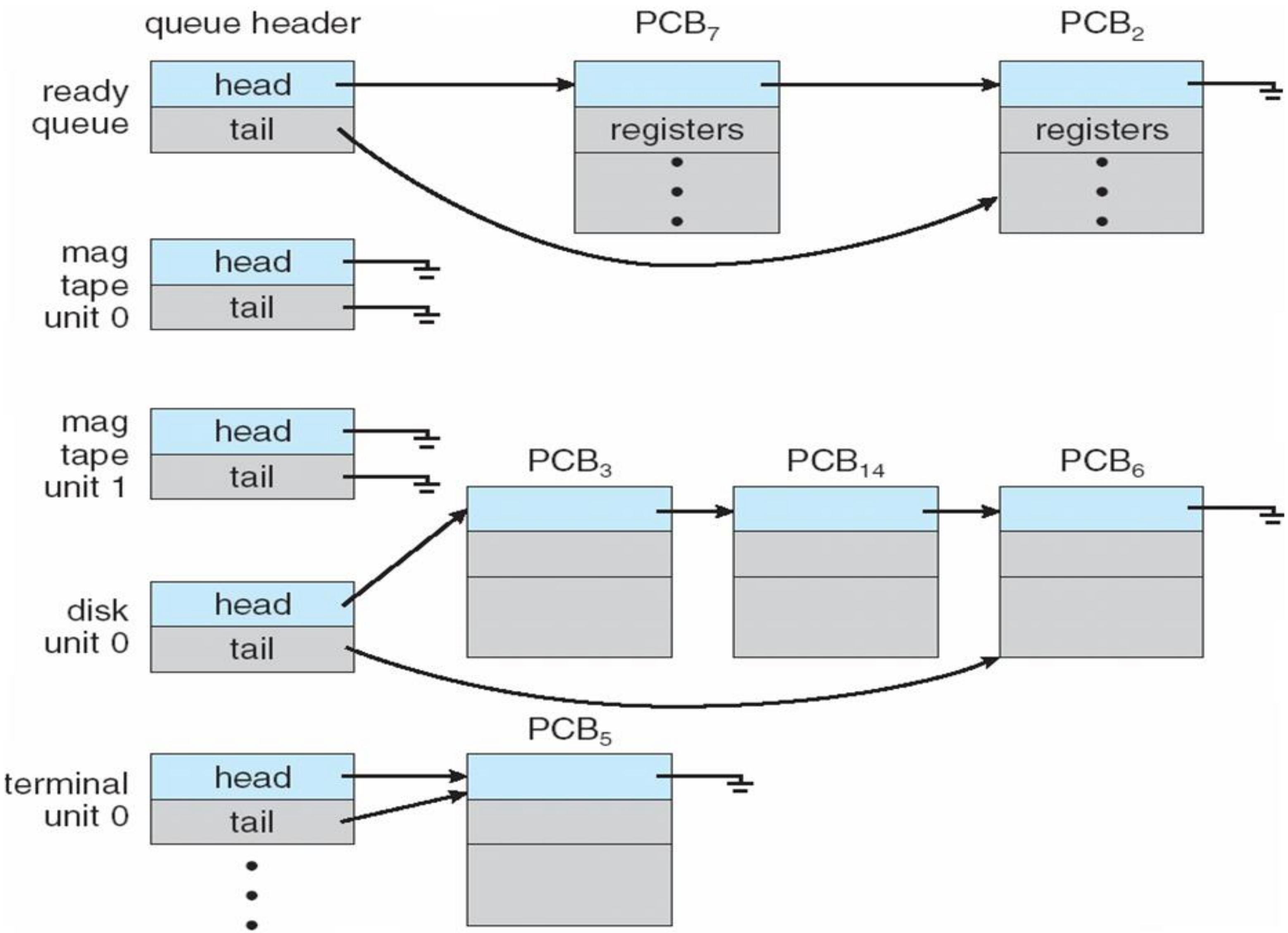
Process Scheduling

- To maximize CPU use, process switch must be fast!
- Process switch AKA context switch
- What is a fast context switch?
  - 1/10 of a second?
  - 1/100 of a second?
  - 1/1000 of a second? (AKA ms)
  - 1/10,000 of a second?
- In one single ms some million instructions (absent memory-access and file IO) can be executed
- So a context-switch time as long as a ~ms seems long, seems to be high overhead
- Typical times: ~100 ns to few µ-s

Process Scheduling

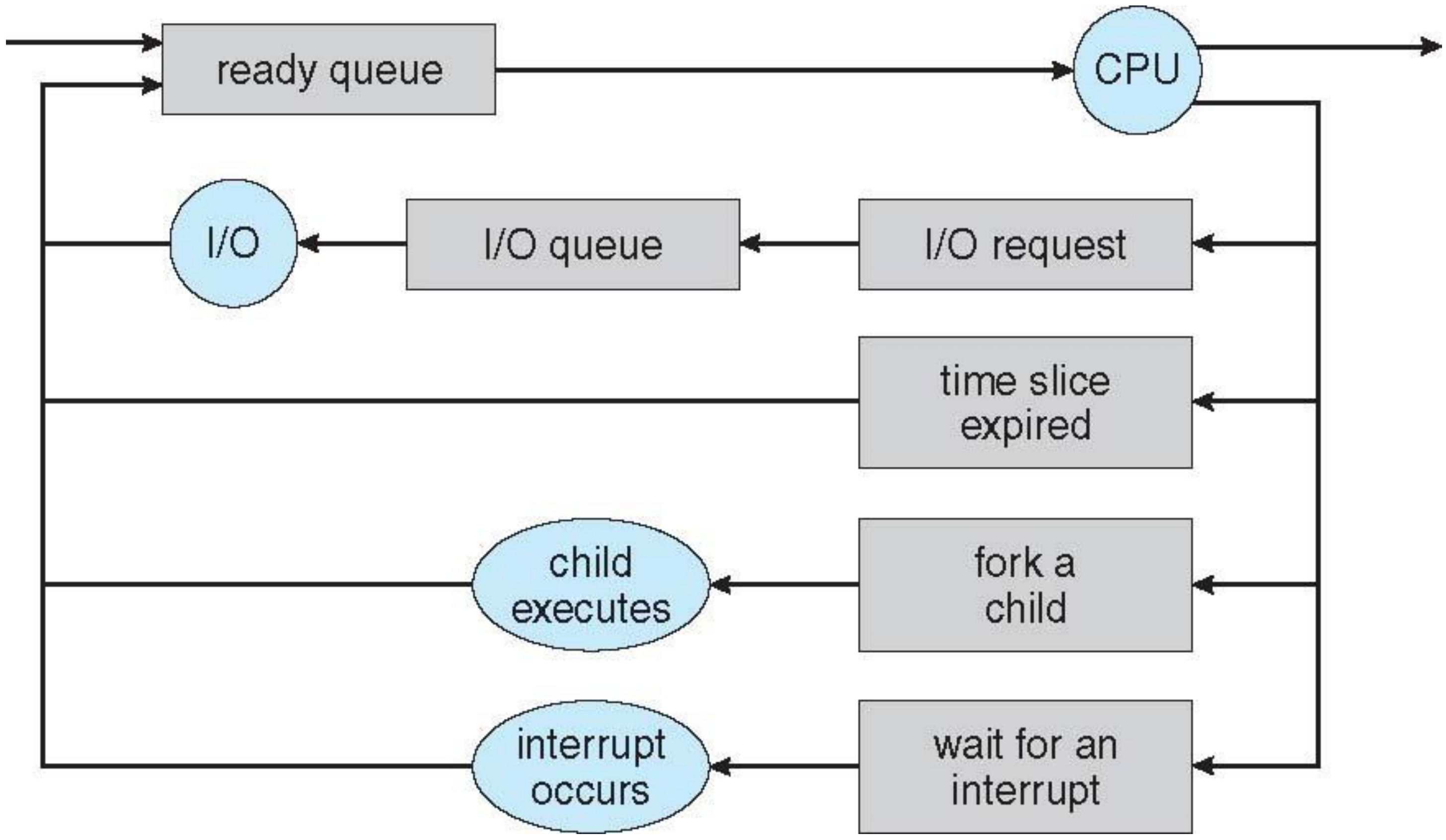
- **Process scheduler (part of OS) selects via interrupt one among several available processes as next candidate for execution –for up to a full time slice; or less if being interrupted again**
- **OS maintains scheduling queues of processes**
  - **Job queue – all processes not yet completed in system**
  - **Ready queue – queue of all processes in main memory, ready and waiting to execute; just need a CPU time slice**
  - **Device queues – processes waiting for device, e.g. for I/O; AKA wait queues**
  - **Other queues as needed**
  - **Processes generally migrate among various queues, and may change priorities, during migration and even while waiting!**

Process Queue Scheduling



Process Scheduling

Queuing diagram representing: queues, resources, flows

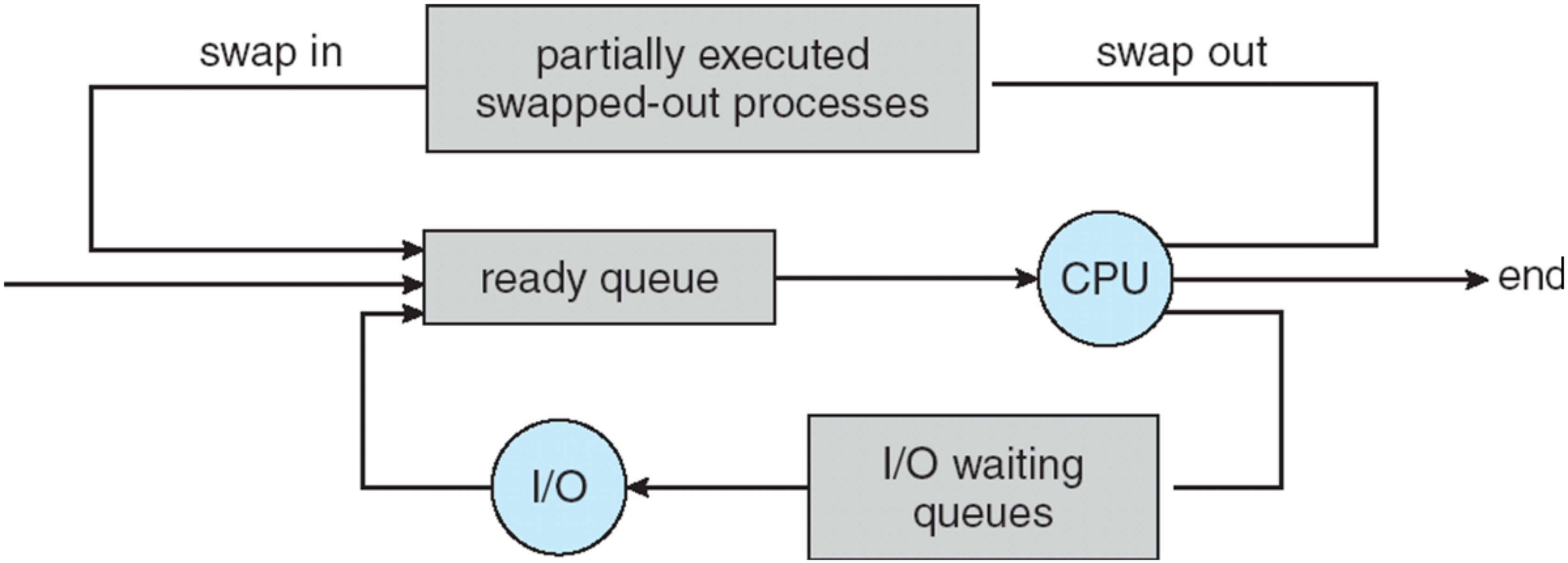


Schedulers

- **1. Short-term scheduler (or CPU scheduler) – selects via interrupt which process to execute next and to grant the CPU to**
  - **Sometimes this is the only scheduler in a system**
  - **Short-term scheduler is invoked frequently: $\Rightarrow$ fast**
- **2. Long-term scheduler (or job scheduler) – selects which processes, and how many, should be brought into ready queue**
  - **Long-term scheduler invoked rarely: $\Rightarrow$ slow OK**
  - **Long-term scheduler controls degree of multiprogramming**
- **Processes can be characterized as:**
  - **I/O-bound process – spends more actual time doing I/O than computations: many short CPU bursts**
  - **CPU-bound process – spends more time doing computations; typical for long CPU bursts**
- **Long-term scheduler strives for good *process mix***

Schedulers

- **3. Medium-term scheduler optional; can be added if degree of multi programming needs to decrease**
- **Removes process from memory, stores on disk, swaps back in from disk to continue execution if situation warrants: swapping**



Multitasking in Mobile Systems

- Some mobile systems –early version of iOS– allow only one process to run (on Apple); others stay suspended, i.e. not ready!
- Due to screen real estate, user interface poses hard limits on mobile platform: iOS only provides for
  - Single foreground process– controlled via user interface
  - Multiple background processes– in memory, running, but not being displayed, and with limits
  - Limits include single, short task, receiving notification of events, specific long-running tasks like audio playback
- Android runs foreground and background

Context Switch

- **When CPU switches to another process, OS must save state of old process and load the saved state of new process: AKA context switch**
- **Context of any process is held in its respective PCB**
- **Context-switch time is overhead: system does no useful user work while switching (is OS bureaucracy ☺)**
  - **Except enforces fairness among users**
  - **Enables higher throughput**
  - **Complex PCB causes slow context switch! Hence: Simplify!**
- **Overhead time, AKA wait time, depends on HW support:**
  - **Some HW provides multiple register sets ➔ multiple contexts loaded simultaneously with live data; but only one running**
  - **e.g. Intel hyperthreaded MPs**
  - **OS can leave data of interrupted process in memory; inactive, but quick to save (~0 cycles) and to resume (~0 cycles)**

Context Switch

Running process will eventually be interrupted!

Processes

Operations on Processes

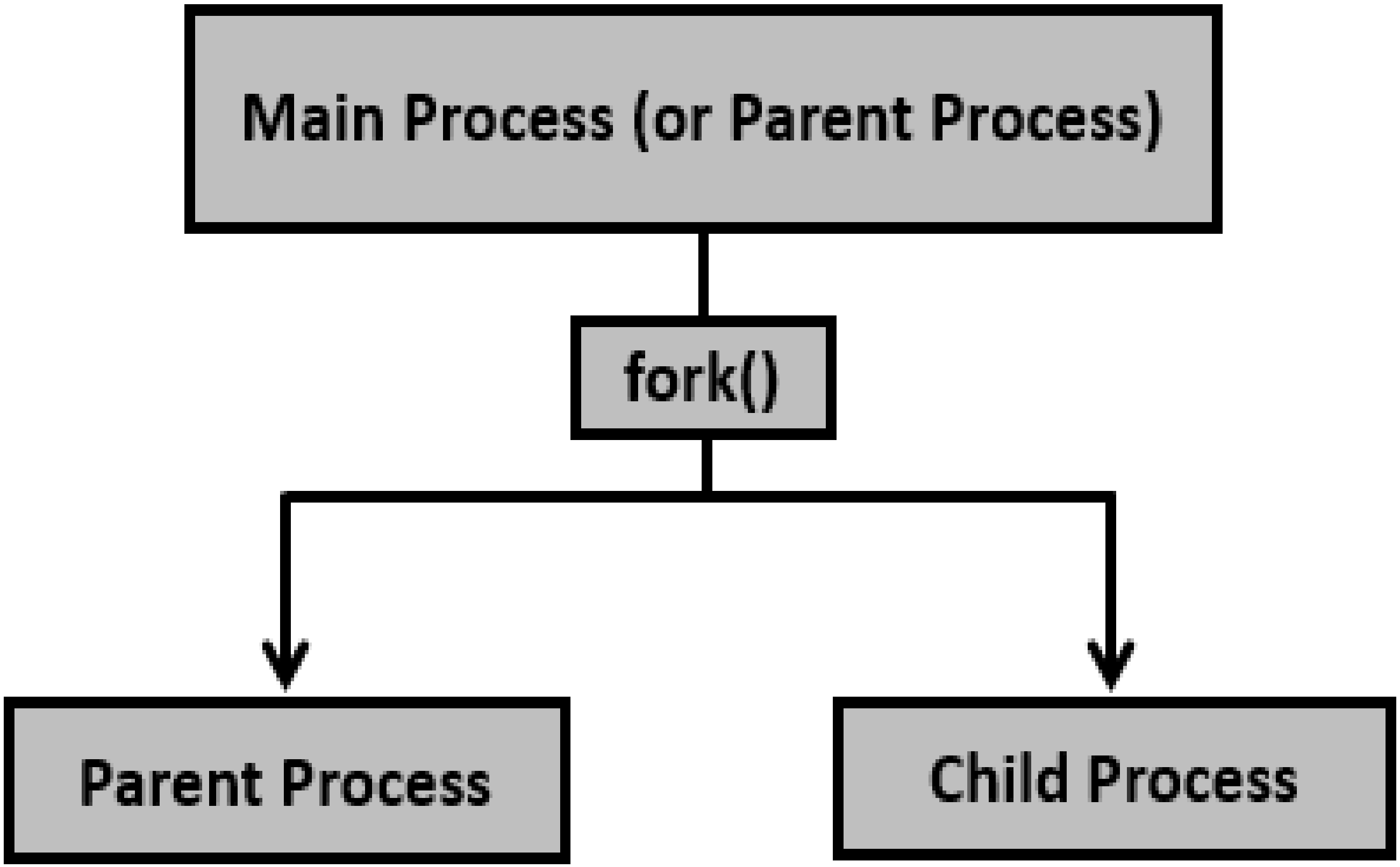
- OS provides mechanism for:
  1. Process creation
  2. Process termination for various normal and abnormal reasons
  3. Programmed process termination via exit() call
  4. Process status to be returned to place of call
  5. Also needed: wait() to ensure clean end-of-life of child
- Must happen concurrently with other processes on the same system, using this single CPU
- Effectively competing (time sharing) for key resources: services + CPU time, *the* critical resource

Process Creation

- Parent process creates child processes via fork(), which, in turn becomes a new, different process via exec(), forming possibly a tree of processes
- Generally, process identified and managed via pid, short for: process identifier; all pids are unique!
- Diverse resource sharing options; OS-dependent, e.g.:
  - Parent and children can share all resources
  - Children at times share only subset of parents' resources
  - Or parent and child share no resources
  - See also: Capability-based OS model, [5]
- Execution order
  - Parent and children execute concurrently
  - Parent waits until children terminate; else zombie processes

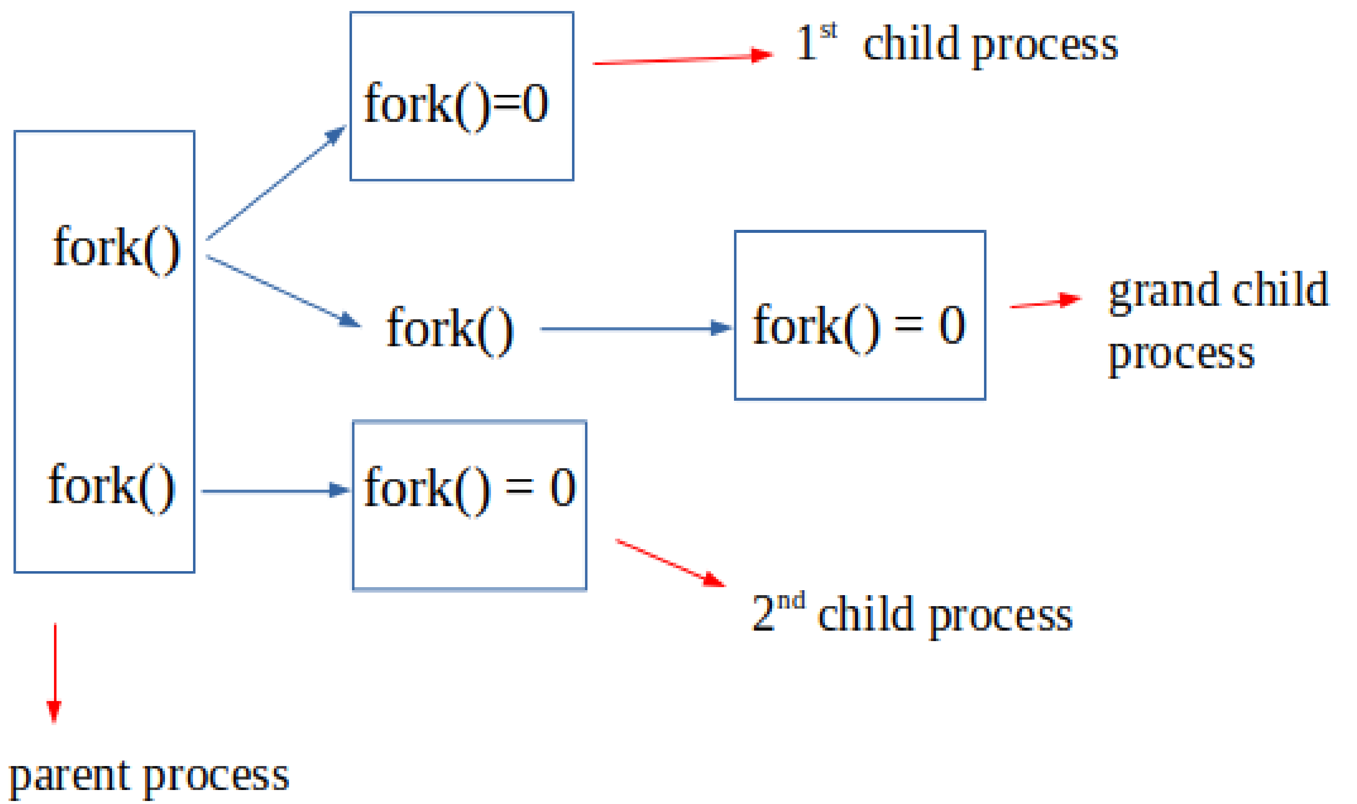
Operations on Processes

Process Creation via fork() on Linux & Unix



Operations on Processes

One Program Spawning Multiple Processes via fork()



`fork()` Processes

- **fork()** **creates a new process by duplicating the calling process; new process is referred to as the *child* process**
- **The one originating the fork() is referred to as the *parent* process**
- **Since parent + child object codes are identical and not modifiable, no need to duplicate!**
- **At fork() time, both memory spaces have identical content –for some time:**
- **Child process is –until exec() call– an exact code duplicate of parent process, except for:**

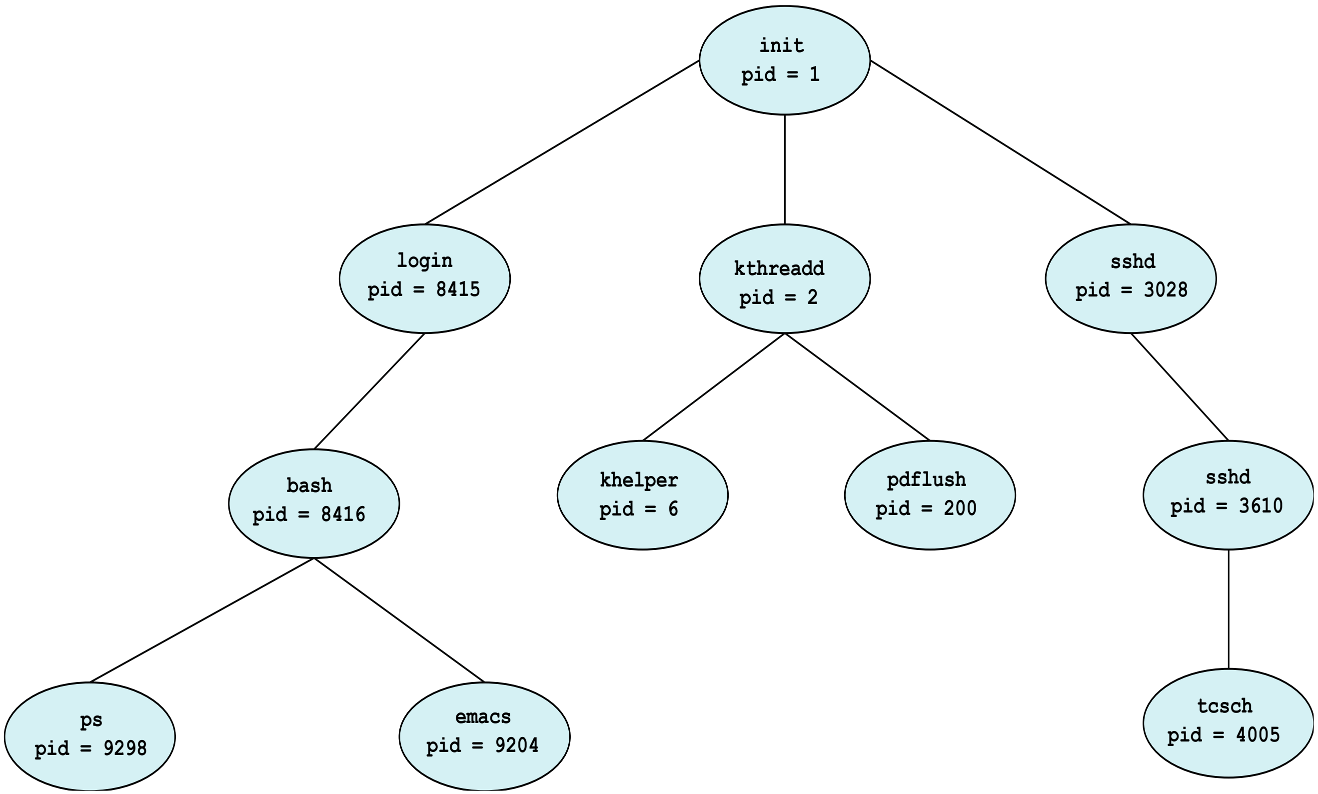
`fork()` Processes

1. **Child has its own unique process identifier (PID), and this PID does not match the ID of any existing process group (*setpgid(2)*) or session**
2. **The child's parent's process ID is simply the parent process ID (PPID)**
3. **The child does not inherit its parent's memory locks (mlock(2), mlockall(2))**
4. **Process resource use (getrusage(2)) and CPU time counters (times(2)) are set to zero in child**
5. **The child's set of pending signals is initially empty (sigpending(2))**

`fork()` Processes

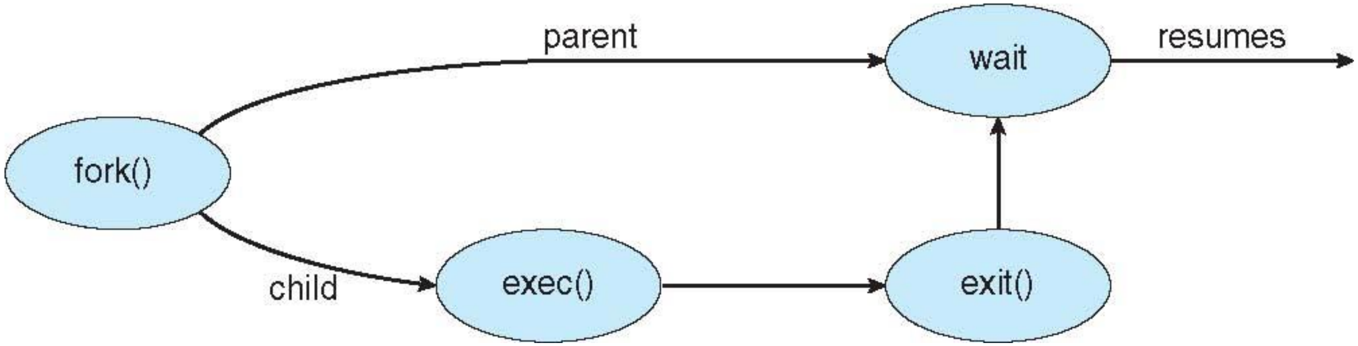
6. **Child does *not* inherit semaphore adjustments from its parent; see [6] on Linux; not covered here**
7. **Child does *not* inherit process-associated record locks from its parent, but does inherit open file description locks from parent**
8. **Child does *not* inherit timers from parent**
9. **Child does *not* inherit outstanding asynchronous I/O operations from its parent, nor asynchronous I/O contexts from parent**

A Tree of Processes in Linux



Process Creation Detail

- Address space
  - Child is duplicate of parent (modulo exceptions listed above)
  - Child can load new, different code into it's space via `exec()`
- UNIX examples
  - `fork()` system call creates new process (a duplicate)
  - `exec()` system call is used after `fork()` call to replace process' memory space with a new program space
  - Also `execve()` common on Linux



Process Termination

- Process executes last instruction, then asks OS to terminate it using `exit()` system call
  - Return status from child to parent is passed via `wait()`
  - Process resources are de-allocated by OS
- Parent may also terminate child processes prematurely, using `abort()` system call; abnormal!
- Plausible reasons for early abort:
  - Child has exceeded allocated resources
  - The task assigned to child is no longer required
  - Parent is exiting for external reasons, and the OS prohibits a child to continue if parent is terminated

Process Termination

- **Some Operating Systems don't allow child to exist if parent terminates: consequently, when process terminates, all its children must terminate as well**
  - **AKA Cascading termination**
  - **All children, grandchildren, etc. are then terminated**
  - **Termination initiated by OS**
- **Parent process may chose to wait for termination of a child process by using the `wait()` system call**

```
pid = wait( & status );
```

Process Termination

- A zombie is in existence, when a parent process does not use wait() after a child terminates to read that child's exit status; i.e. there is no record that the child ever did die!
- Zombie process exists via an entry in the OS process table; i.e. zombie is a process in terminated state, but is still "accounted for"
- An orphan is a child process whose original parent process terminates before that child process itself terminates
- Detail:

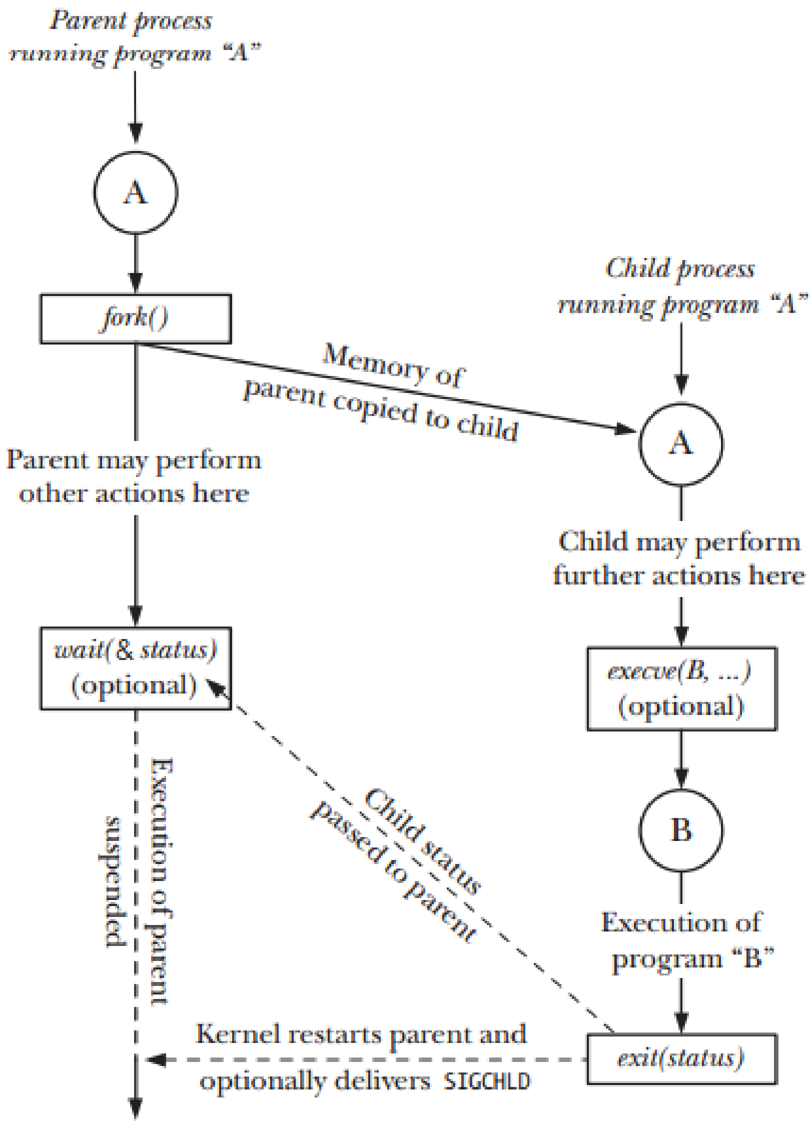
Process Termination

- **Under Unix, a zombie process or a defunct process is a process that has completed execution via exit() but still has an entry in the process table**
- **A zombie is a process in the Terminated state**
- **This occurs, whenever the entry in the process table is needed to allow the parent to read its child's exit status**
- **This is generally an orderly way of executing!**
- **Once the exit status is read via wait(), the zombie's entry is removed from the process table**
- **Then the zombie is said to be *"reaped"***

Process Termination

- A child process first becomes a zombie before being removed from the resource table
- Generally, zombies are immediately waited on by their parent and then *reaped* by the system
- Processes that stay zombies for a long time are generally an error and cause a minor resource leak
- But the only true resource they consume is the one entry in the process table

Process Termination



Unix

- Contemporary **process model** was popularized under Unix
- Key contributors: Dennis **Richie** and Ken **Thompson**
- Here hard ☺ at work on a PDP11 in 1972
- Refining Unix OS
- **Process concept** had already been developed way before Unix

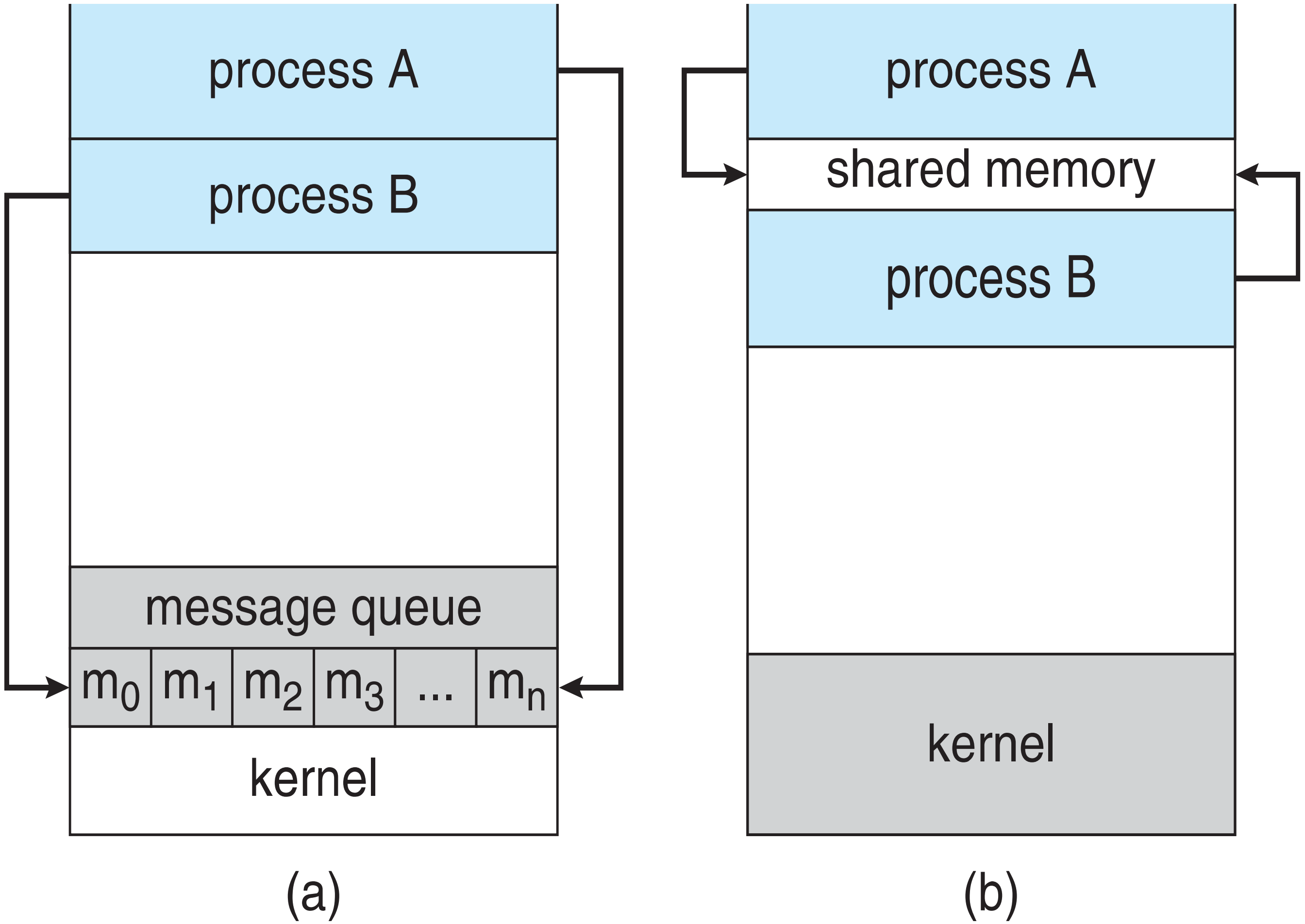
IPC via Shared Memory

IPC Interprocess Communication

- **Processes may be *independent* or *cooperating***
- **Cooperating ones can affect each other**
- **Reasons for cooperating processes:**
  - **Information sharing, e.g. File sharing**
  - **Computation speedup**
  - **Speedup even on UP; if otherwise parent would have to wait for some event to continue computing**
- **Cooperating processes need so called interprocess communication, IPC**
- **Two communication models of IPC**
  - **Message Passing**
  - **Shared Memory**

Communication Models

a) Message Passing b) Shared Memory



Cooperating Processes

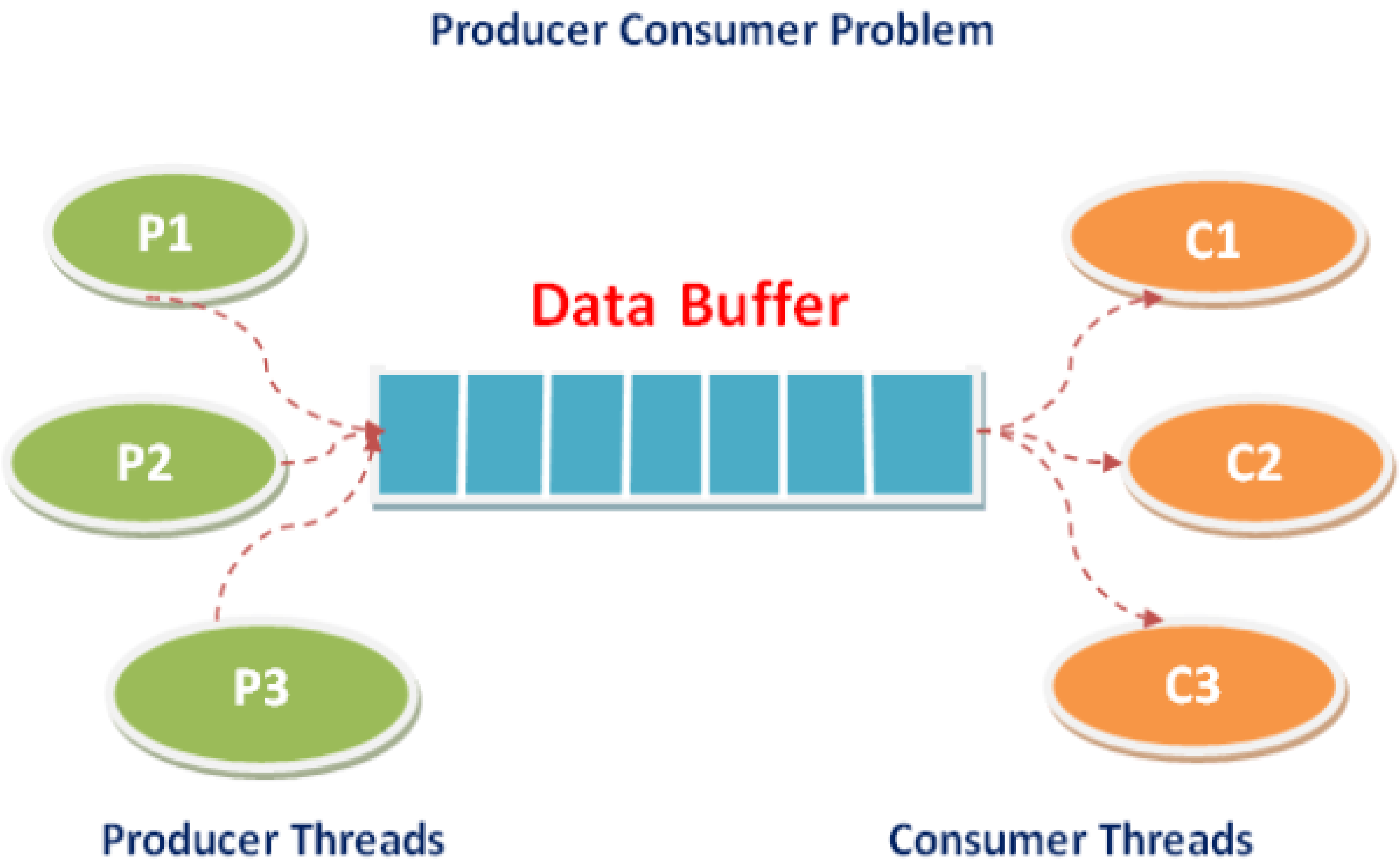
- ***Independent*** **process cannot affect, or be affected by, execution of another process . . .**
- **Other than having to share a key resource: CPU**
- ***Cooperating*** **process can affect, or be affected by, the execution of another process, the one it cooperates with**

Producer Consumer Problem

- **Paradigm for cooperating processes, AKA the so called *producer consumer problem*:**
- ***Producer* process generates information, placed into non-full buffer**
  - **Note: Producer must wait() when buffer is *already full!***
- ***Consumer* process absorbs such information from a non-empty buffer**
  - **Note: Consumer must wait() when buffer is *still empty!***
- **Unbounded Buffer places no hard, practical limit on the buffer size; hard to implement ☺ needs infinite space**
- **Bounded Buffer ”knows” there is a fixed buffer size; can cause wait() for either producer or consumer**

Producer Consumer Problem

Paradigm for Sharing Data Cooperatively



The Bounded Buffer Problem

Bounded Buffer, Shared-Memory

- **Bounded Buffer paradigm involves processes, and:**
  - **A *shared data structure*, AKA the buffer[ ]**
  - **Bounded buffer problem employs producer and consumer actions. Producer and consumer . . .**
  - **. . . need data structures: in and out, indexing next element to be produced, and next element to be consumed in buffer[ ]**
  - **Producer places information into shared buffer[in]**
  - **Consumer removes information form shared buffer[out]**
- **Consumer has to wait, when buffer is still empty**
- **Producer has to wait, when buffer is already full**
- **In and out are checked and “synchronized” (updated) modulo buffer size**

Bounded Buffer, Shared-Memory

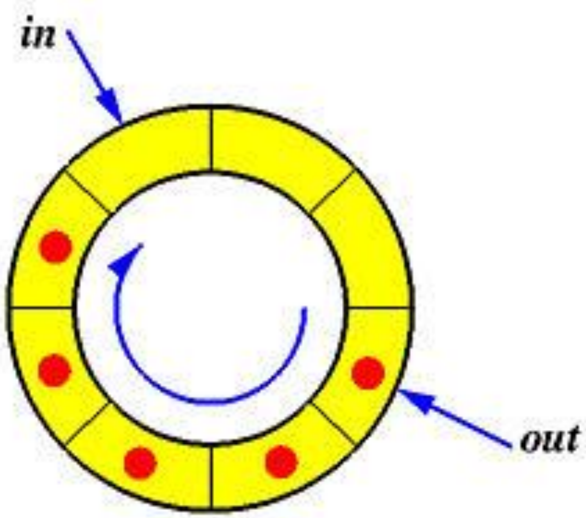
- **Shared data:**

```
#define BUFFER_SIZE 10        // shared size e.g. 10
typedef struct {              // type of buffer elem.
          . . .               // real data here!
    } item_tp;                // arbitrarily complex
item_tp buffer[ BUFFER_SIZE ]; // shared
int  in = 0;             // should be subrange; shared
int out = 0;             // also an "index"; shared
```

- **Solution is correct, but has a limited, fixed number of BUFFER_SIZE - 1 elements; with loss of 1 index!**
- **Must reserve one combination –one lost index value– for condition: Buffer is empty! Index of the "wasted" slot doesn't have to be 0! Generally isn't, only initially**

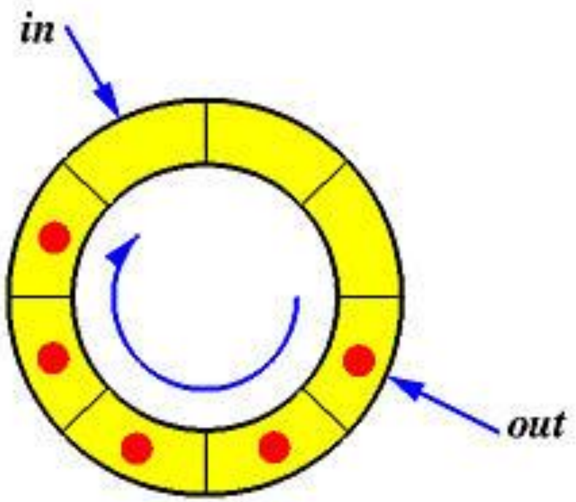
Bounded Buffer Producer

```
item_tp next_produced;        // local to producer
while( true ) {               // do forever! w. interrupt
      // generate next_produced, place in buffer[ in ]
      next_produced = ...     // Real work here: produce!
      // now check: space in buffer[]?
      while( ( ( in + 1 ) % BUFFER_SIZE ) == out ) {
            ; // wait; full buffer
      } //end while
      buffer[ in ] = next_produced;
      // only producer sets: in
      in = ( in + 1 ) % BUFFER_SIZE;
} //end while
```



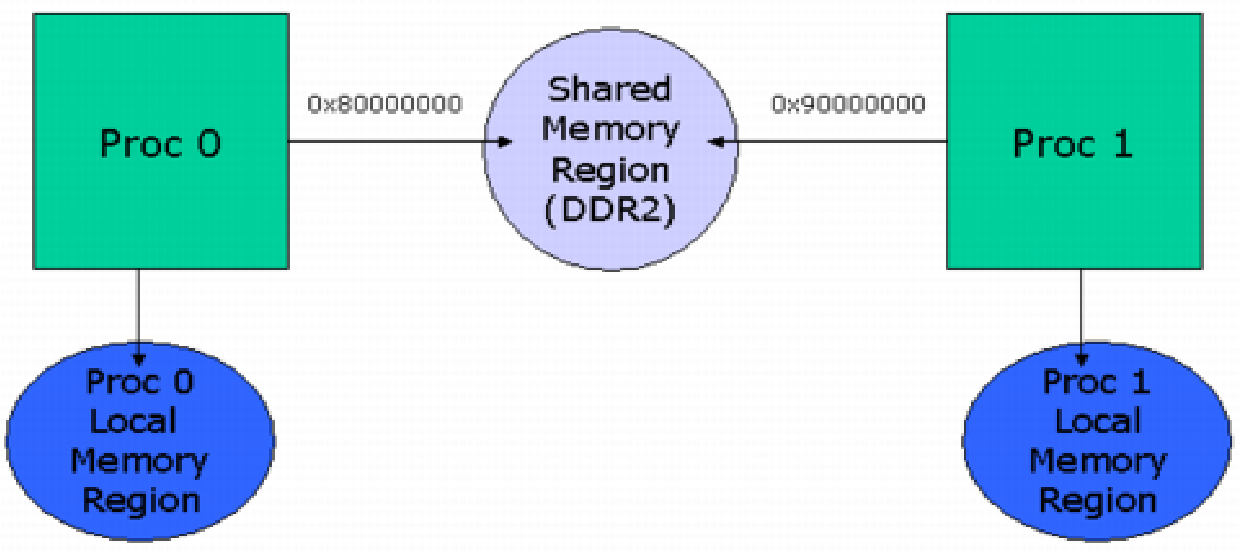
Bounded Buffer Consumer

```
item_tp next_consumed;               // local to consumer
while( true ){                       // do forever
      while( in == out ) {           // empty buffer[]?
            ; // wait                // nothing to consume
      } //end while
      // next element available at buffer[ out ]
      next_consumed = buffer[ out ];
      // only consumer sets: out
      out = ( out + 1 ) % BUFFER_SIZE
      // consume next_consumed
      // Real work here: consume!
      . . . = next_consumed
} //end while
```



IPC Shared Memory

- An area of memory shared among processes that need to communicate, occasionally!
- Communication under control of user processes, not under OS control!
- Major goal: Provide a mechanism to allow user processes to synchronize their actions when accessing shared memory



Interprocess Communication (IPC) via Message Passing

IPC Message Passing

- IPC is a mechanism for processes to communicate and to synchronize their actions, when necessary! Done by sending a message; use a message system!
- Generally they progress independently! Yet at specific times, they must cooperate and coordinate!
- Message system means: Processes communicate with each other without shared objects
- IPC facility provides two message operations:
  - `send( message )`
  - `receive( message )`
- *Message* size can be fixed or variable

IPC Message Passing

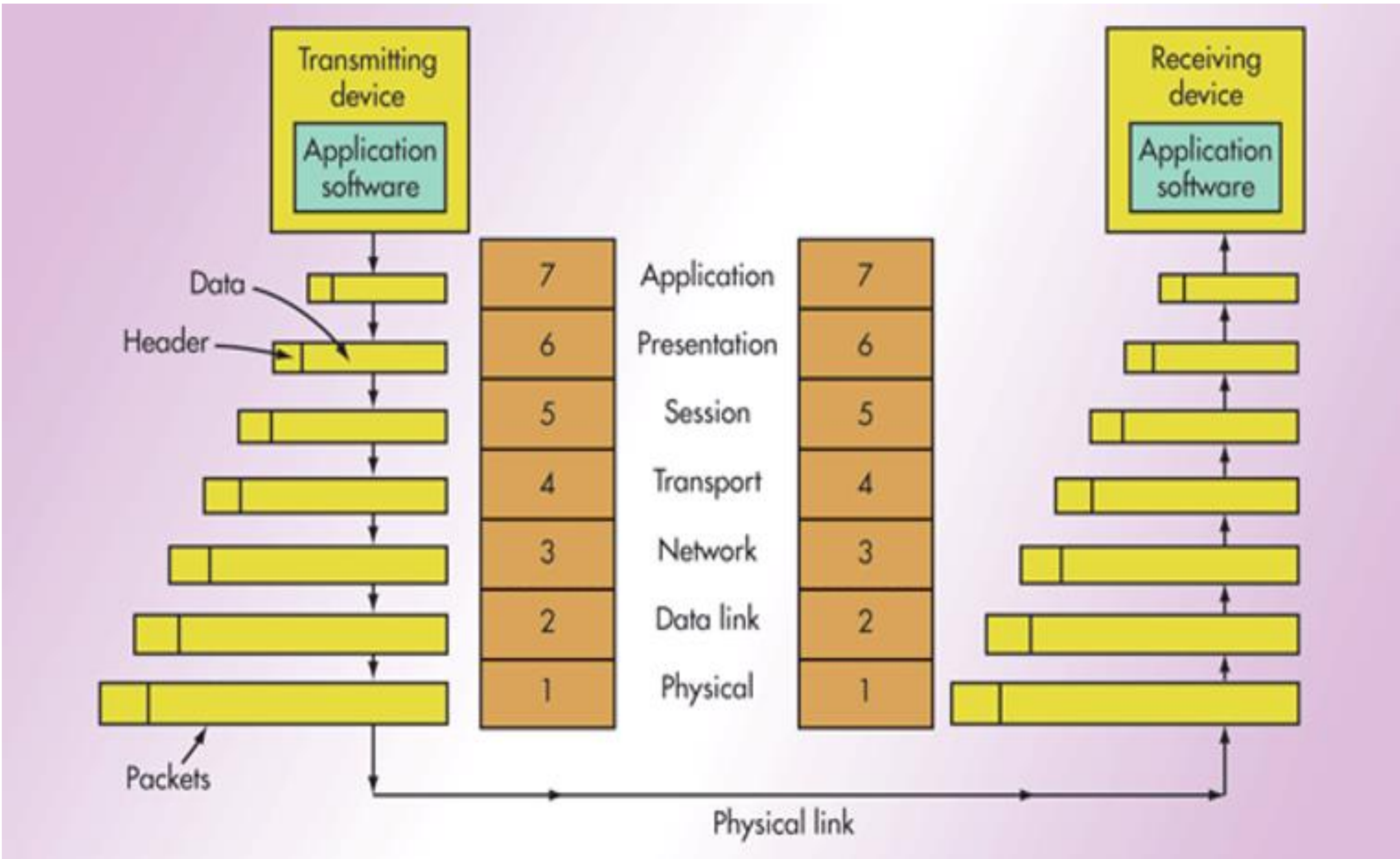
- **For processes *P* and *Q* to communicate, they need to:**
  - **Establish a *communication link* between them**
  - **Exchange messages via send() & receive()**
- **Implementation issues and OS design parameters:**
  - **How are links established?**
  - **Can a link be associated with more than just two processes?**
  - **How many links can there be, between any pair of communicating processes?**
  - **What is the capacity of a link?**
  - **Is the message size fixed or variable?**
  - **Is a link unidirectional or bi-directional?**

IPC Message Passing

- **Communication link** is a **channel** connecting two or more communicating devices
- **May be an physical link or a logical link that uses some other physical connection**
- **Implementation of communication link:**
  - **Physical:**
    - **Special HW bus**
    - **Network**
    - **Others**
  - **Logical:**
    - **Direct or indirect communication**
    - **Synchronous or asynchronous**
    - **Automatic or explicit buffering**
- **Reminders TCP/IP: Transmission Control Protocol, internet protocol. IPC: Inter-Process Communication**

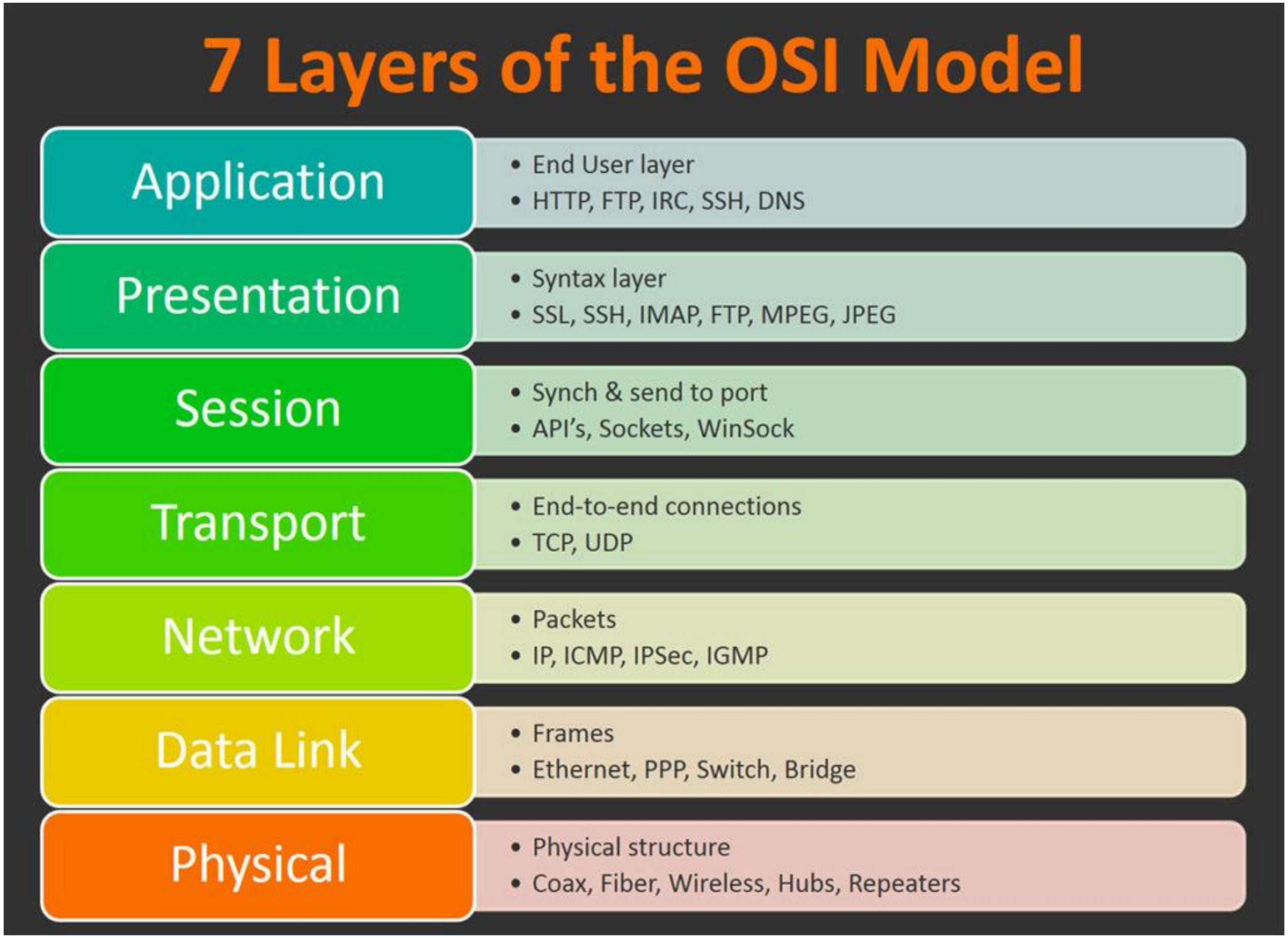
IPC Message Passing

Open Systems Interconnect (OSI): the 7 Layers



IPC Message Passing

Different Image, Same 7 Layer OSI Model



Direct & Indirect Communication

Direct Communication

Direct Communication

- Processes must name each other explicitly in direct communication:
  - `send( P, message )` – send message to process P
  - `receive( Q, message )` – receive message from process Q
- Properties of communication link
  - Links are established automatically
  - A link is associated with exactly one pair of communicating processes
  - Between each pair there exists exactly one link
  - Link may be unidirectional, but is usually bi-directional

Indirect Communication

- Indirect communication involves a mailbox
- Mailbox being an external device, NOT part of either process!
- But all involved processes can access such a mailbox
- Web on indirect communication: http://people.scs.carleton.ca/~maheshwa/courses/300/l4/node12.html

Indirect Communication

- Messages are directed to, and received from mailboxes (also referred to as ports)
  - Each mailbox has its own unique id
  - Processes can communicate indirectly only if they share such a mailbox
- Properties of indirect communication link
  1. Link established only if processes share a common mailbox
  2. A link may be associated with more than two processes
  3. Each pair of processes may share several communication links
  4. Link may be unidirectional or bi-directional

Indirect Communication

Indirect Communication

- Operations
  - create a new mailbox; or port
  - send and receive messages through mailbox
  - destroy a mailbox
- Primitives are defined as send() receive():

Note A is a mailbox, not a process ID as on p. 70!

`send( A, message )`	send message to mailbox A
`receive( A, message )`	receive message from mailbox A

Indirect Communication

- **Mailbox sharing**
  - P_1, P_2, and P_3 **share** mailbox A
  - P_1 **sends**; P_2 and P_3 **receive**
  - **Which of the two will receive (AKA read) the message?**
- **Solution Options**
  1. **For example, allow a link to be associated with at most two processes; the problem disappears**
  2. **Allow only one process at a time to execute a receive operation; requires mutex protocol**
  3. **Allow the system to select arbitrarily the receiver, and then the sender is notified who the ultimate receiver was**

Indirect Communication

Synchronization

Synchronization

- Message passing may be blocking or non-blocking
- Blocking is considered: synchronous
  - Blocking send: Sender is blocked until the message is received; “Receiver: I can’t read your message now; just wait!”
  - Blocking receive: Receiver is blocked until a message is available; “Sender: I have nothing to send yet; just wait!”
- Non-blocking is considered: asynchronous
  - Non-blocking send: Sender simply sends the message and continues
  - Non-blocking receive: Receiver accepts message
    - A valid message s received, or
    - If no message: continue and check back later, periodically

Synchronization

Synchronization

- **Synchronous serial communication describes a serial communication protocol in which data are sent at a constant rate**
- **Synchronous communication requires that clocks in transmitting and receiving devices be *synchronized* – running at the same rate – so the receiver can sample the signal at the same time intervals used by transmitter**
- **No start or stop bits are required. For this reason synchronous communication permits more information to be passed over a circuit per unit time than asynchronous serial communication**
- **Over time transmitting and receiving clocks tend to drift apart, requiring *resynchronization***

Remote Procedure Call (RPC)

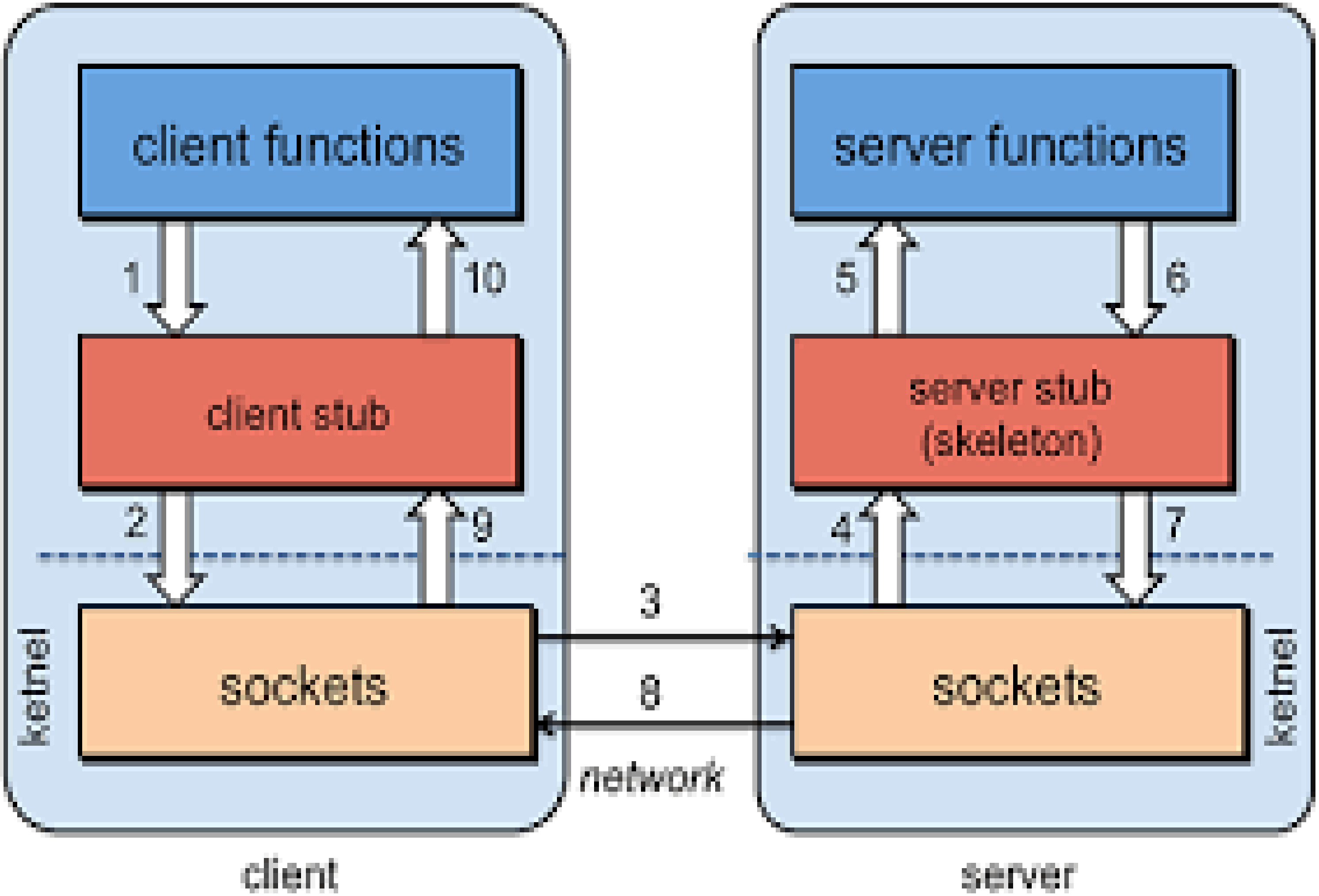
Remote Procedure Call

- Remote Procedure Call (RPC) is a protocol that one program, executing on one computer, can use to request a service from another program located on another computer on some network
- Without having to understand the network's details
- Procedure is AKA function call or subroutine call
- RPC thus executes the client-server model

Remote Procedure Call

- RPC abstracts procedure calls between processes on networked systems
- Stub – is SW that converts parameters passed between client and server during RPC
- Client-side stub locates server and marshals params
- Server-side stub receives message, unpacks the marshaled parameters, and performs the procedure on the server
- Marshaling: Process of transforming the memory representation of an object to another data format suitable for storage or transmission

Remote Procedure Call



Remote Procedure Call

- On Windows, stub codes compile from specification written in Microsoft Interface Definition Language (MIDL)
- Data representation handled via External Data Representation (XDL) format to account for different architectures
    - Big-endian and little-endian
- Remote communication has more failure scenarios than local
    - Messages can be delivered *exactly once* rather than *at most once*
- OS typically provides a rendezvous (or matchmaker) service to connect client and server

Steps of RPC

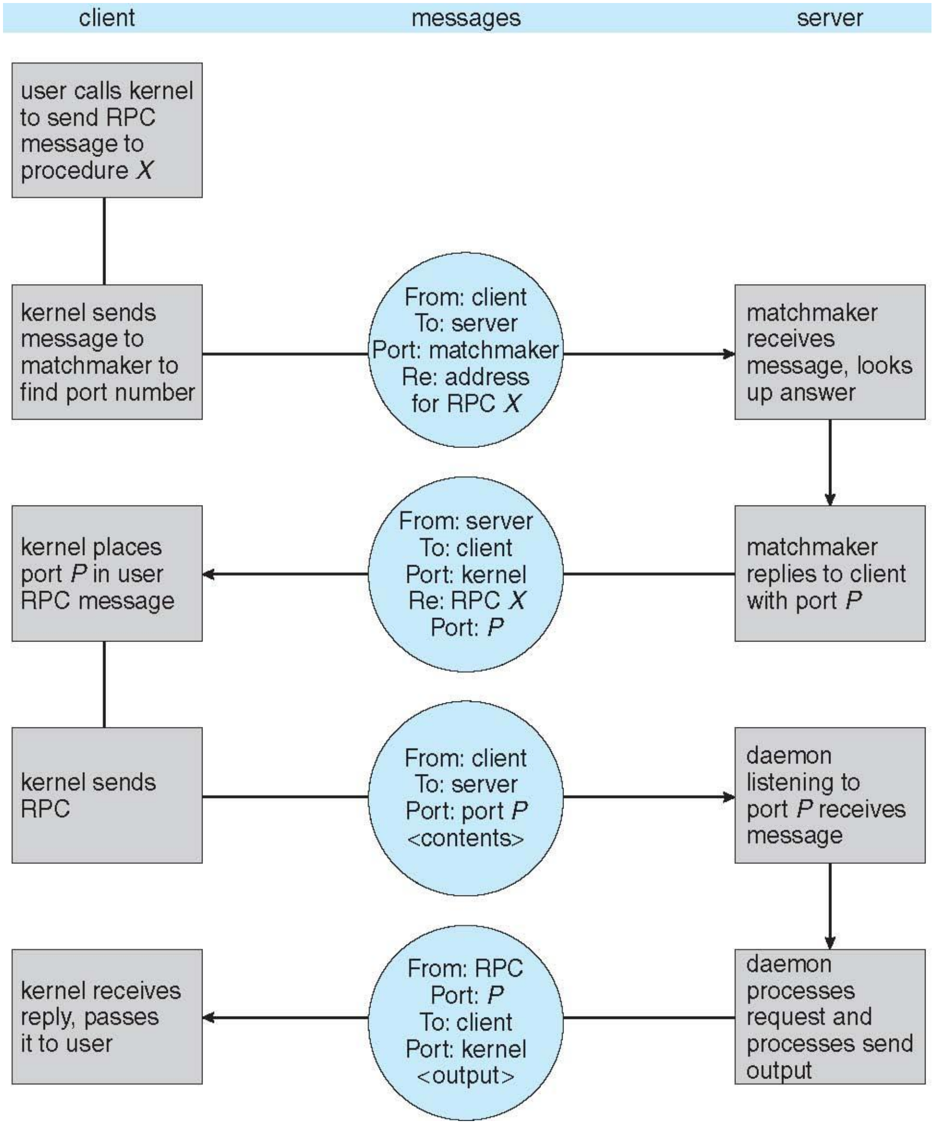
- **Remote procedure call: Interprocess communication technique used for client-server based applications**
- **Client has a request message that the RPC translates and sends to the server**
- **This request may be a procedure or a function call to a remote server**
- **When server receives the request, it sends the required response back to the client**
- **The client is blocked while the server is processing the call; resumes execution after the server is finished**
- **The sequence of 6 defined events in remote procedure call is:**

Steps of RPC

1. Client stub is called by client
2. Client stub makes a system call to send the message to server, places parameters into message
3. Message now is sent from the client to the server by the client OS
4. Message is passed to server stub by the server OS
5. Parameters are removed from the message by server stub
6. Then server procedure is called by the server stub

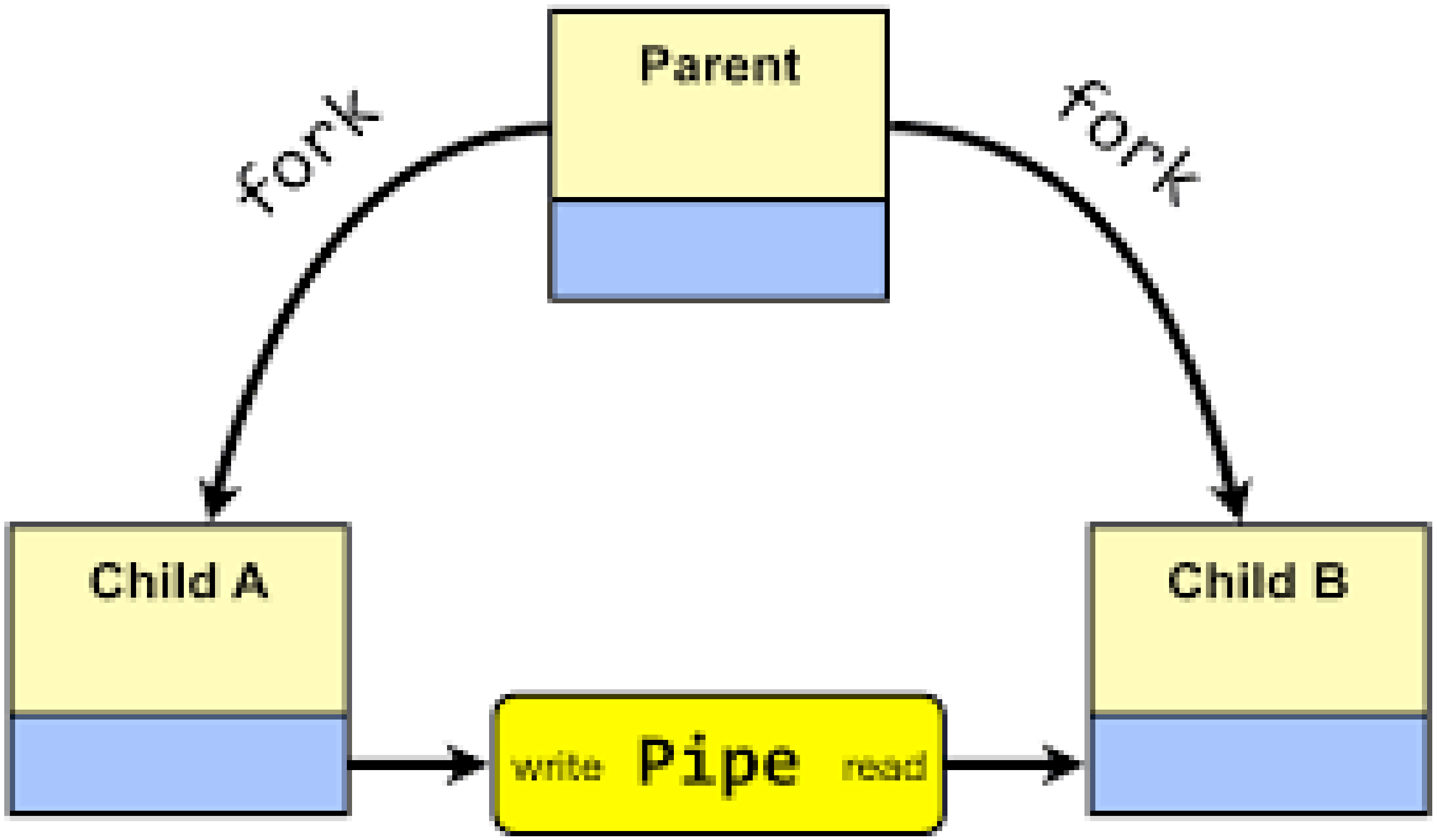
- Note the generic stub-to-OS and OS-to-stub mechanism, which allows "detail" to be handled locally; network has no "need to know"

Execution of RPC



Pipes

Pipes

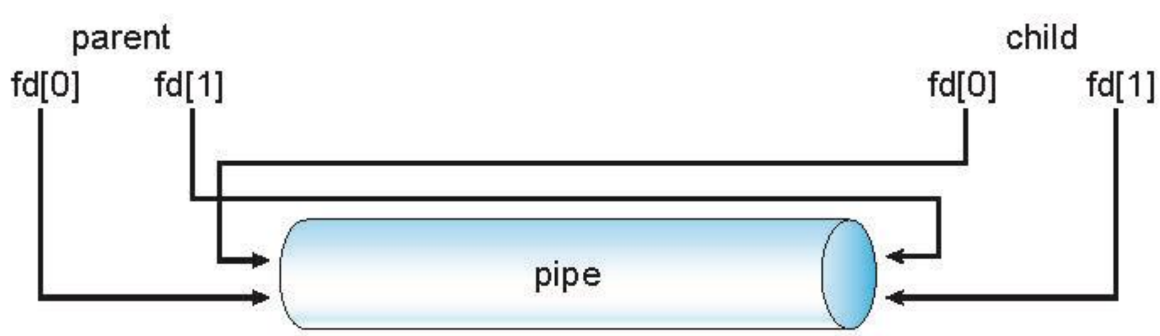


Pipes

- **A Unix Pipe is a conduit allowing two processes to communicate**
- **Design options to be decided:**
  - **Is communication unidirectional or bidirectional?**
  - **In case of two-way communication: half or full-duplex?**
  - **Must there exist a relationship (i.e., *parent-child*) between the communicating processes?**
  - **Can the pipes be used over a network?**
- **Ordinary pipe cannot be accessed from outside process that created it**
- **Parent process creates pipe and uses it to communicate with a child process**
- **Named pipe can be accessed without parent-child relationship**

Ordinary Pipes

- Ordinary Pipes allow communication in standard producer-consumer style
- Producer writes to one end; AKA the write-end
- Consumer reads from other end; AKA read-end
- Ordinary pipes are therefore unidirectional
- Require parent-child relationship between communicating processes
- Windows calls these anonymous pipes



Named Pipes

Named Pipes have added capabilities vs. ordinary pipes:

- **Communication is bidirectional**
- **No parent-child relationship is necessary between the communicating processes**
- **Several processes can use the named pipe for communication**
- **Provided on both UNIX and Windows systems**

Summary

- Process is a program in execution, in one of various states of execution!
- In Unix-like environment, OS tracks processes through five-digit ID known as the pid, short for process ID
- Interprocess communication (IPC) is a programming interfaces to coordinate activities among different processes running concurrently
- Allows program to handle multiple simultaneous user requests
- Remote Procedure Call (RPC) used to request service from a program located in computer on a network
- Pipe is a unidirectional data channel used for interprocess communication

Bibliography

1. Unix process: https://www.tutorialspoint.com/unix/unix-processes.htm
2. Linux process management: https://www.guru99.com/managing-processes-in-linux.html
3. IPC: https://en.wikipedia.org/wiki/Inter-process_communication
4. C data structure for Linux VMM: https://www.tldp.org/LDP/tlk/ds/ds.html
5. Capability Model: https://en.wikipedia.org/wiki/Capability-based_security
6. Semaphore adjustment: http://man7.org/linux/man-pages/man2/semop.2.html