Chapter 3: Process

Zhi Wang
Florida State University

Contents

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

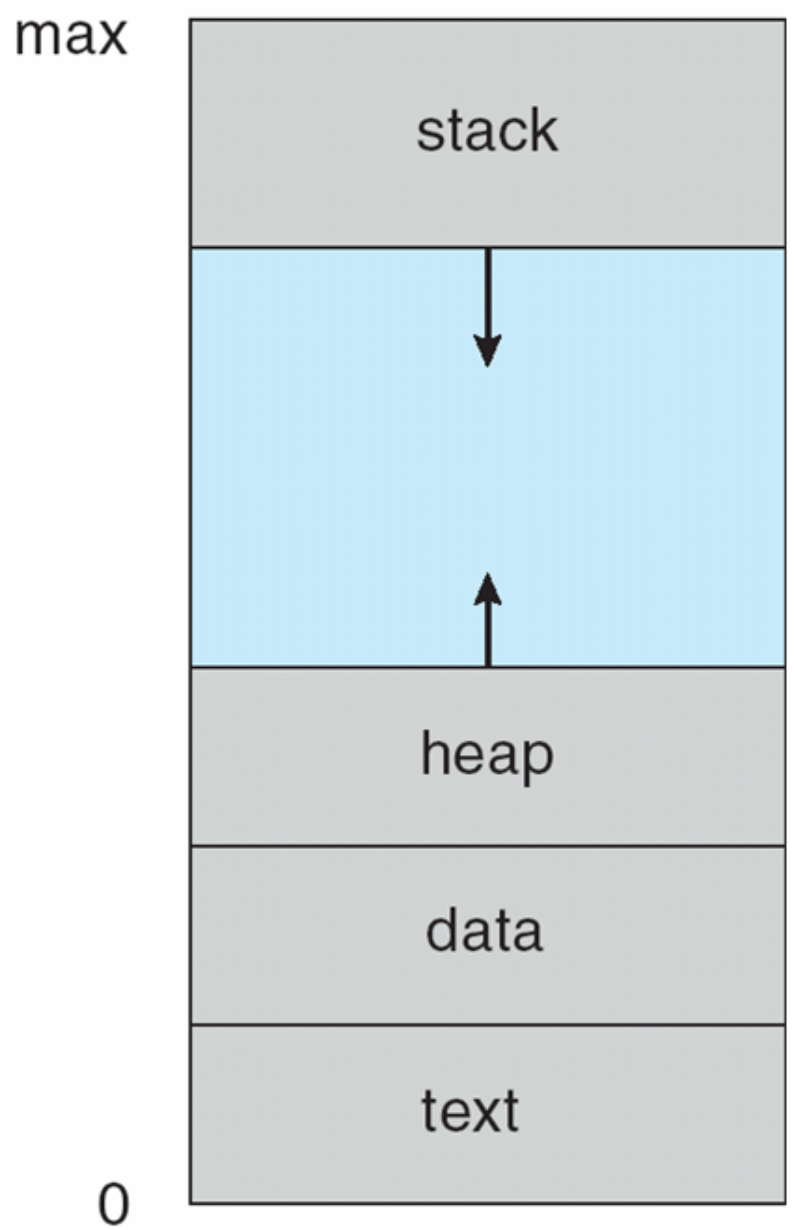
Process Concept

- 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)
  - process can be started via GUI or command line entry of its name, etc

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

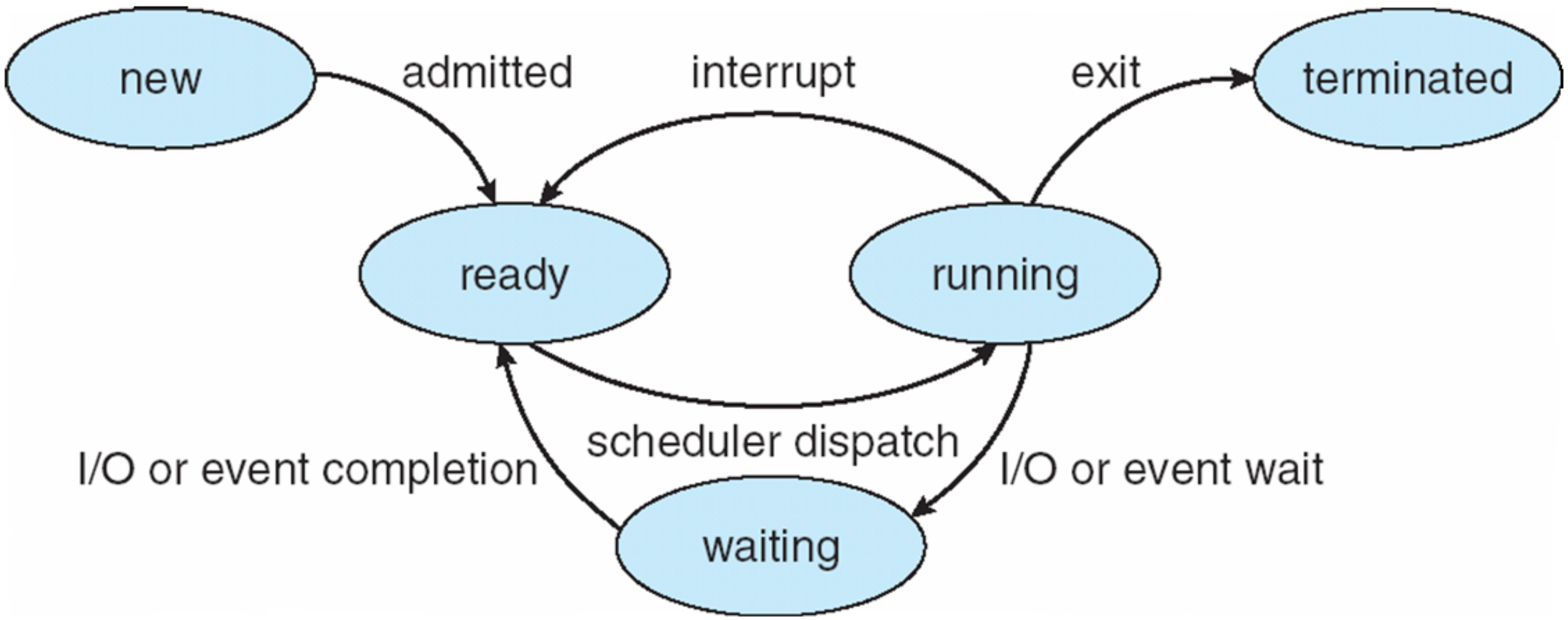
Process in Memory



Process State

- As a process executes, it changes state
  - **new**: the process is being created
  - **running**: instructions are being executed
  - **waiting**: 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

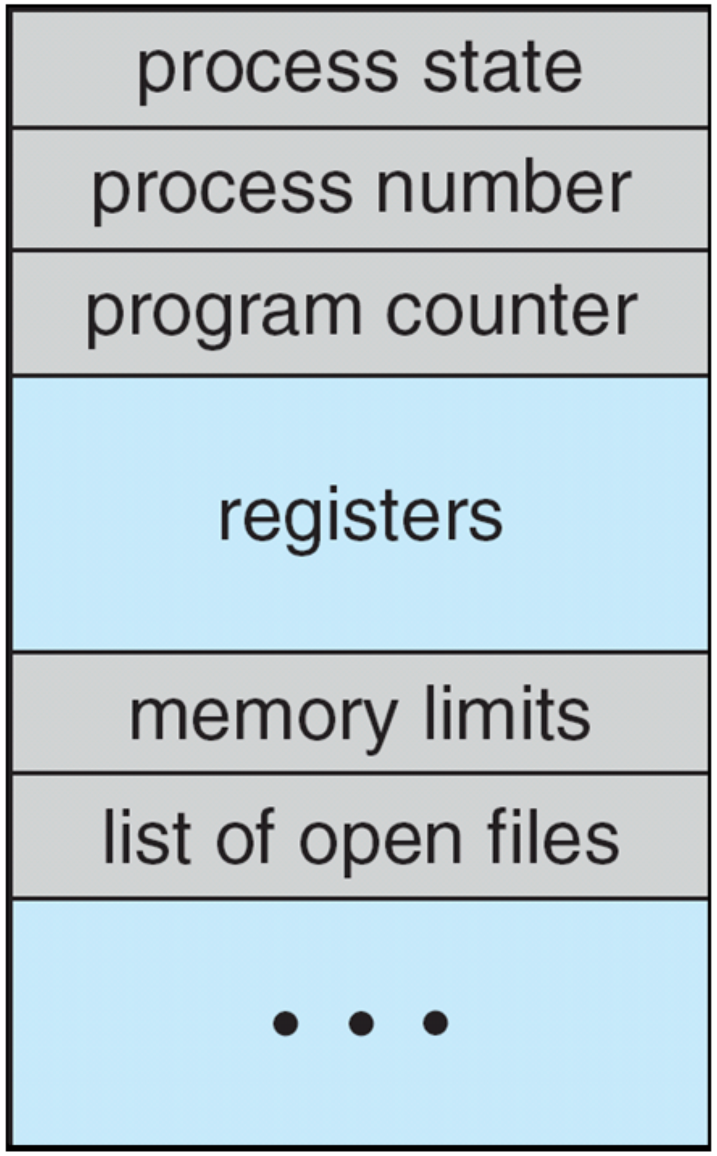
Diagram of Process State



Process Control Block (PCB)

- In the kernel, each process is associated with a **process control block**
  - process number (pid)
  - process state
  - **program counter**
  - 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

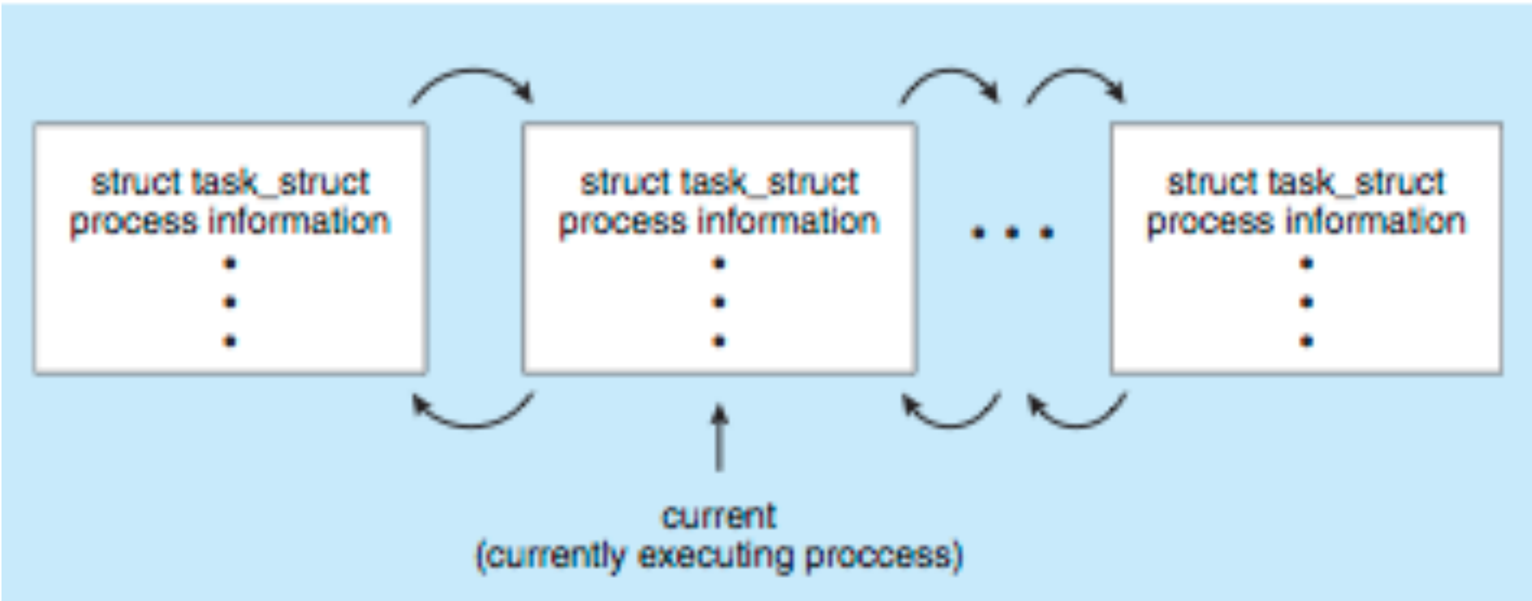
Process Control Block (PCB)



Process Control Block in Linux

- Represented by the C structure **task_struct**

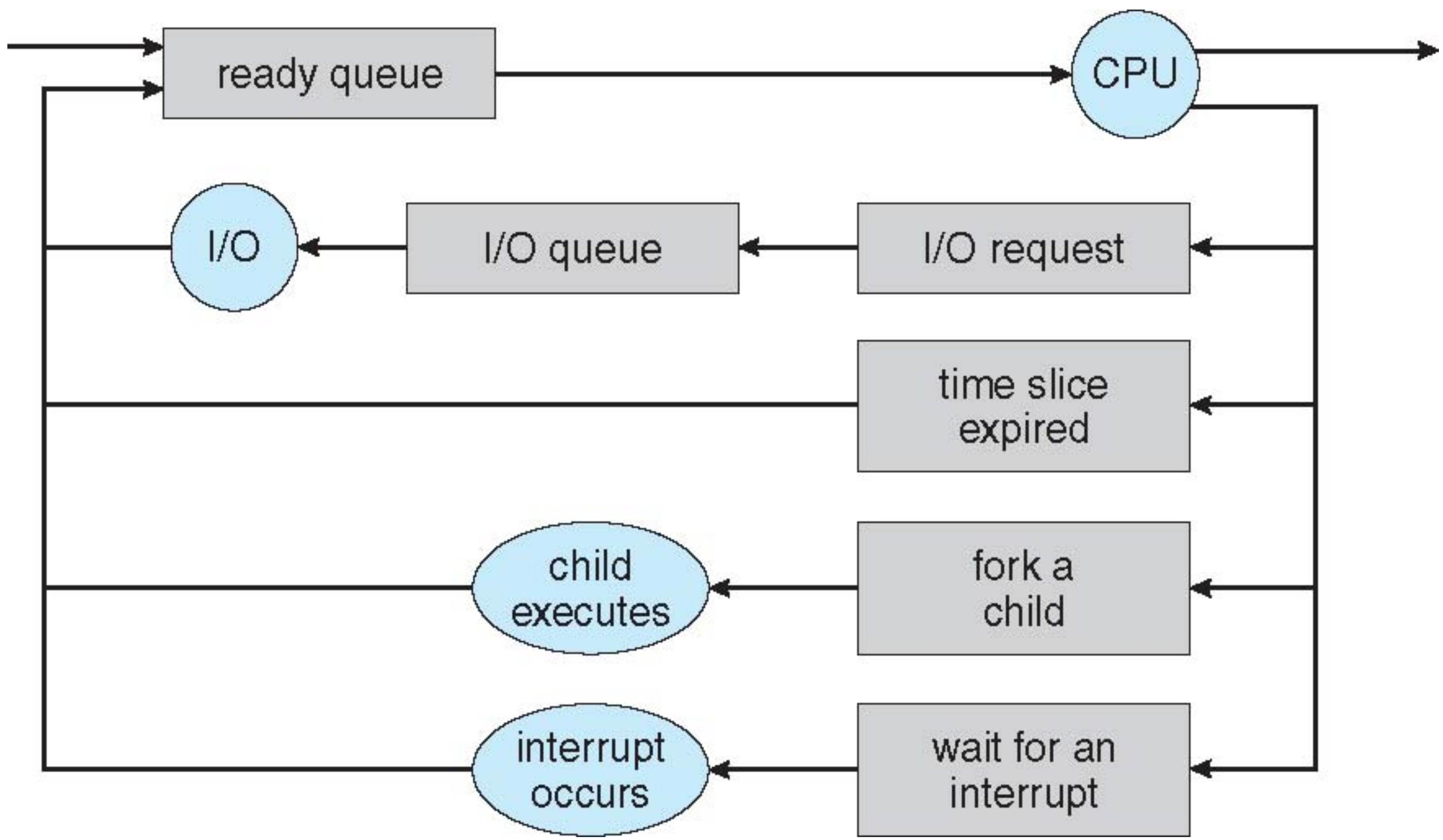
```
pid_t pid;                       /* process identifier */
long state;                      /* state of the process */
unsigned int time_slice          /* scheduling information */
struct task struct *parent;      /* this process's parent */
struct list head children;       /* this process's children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



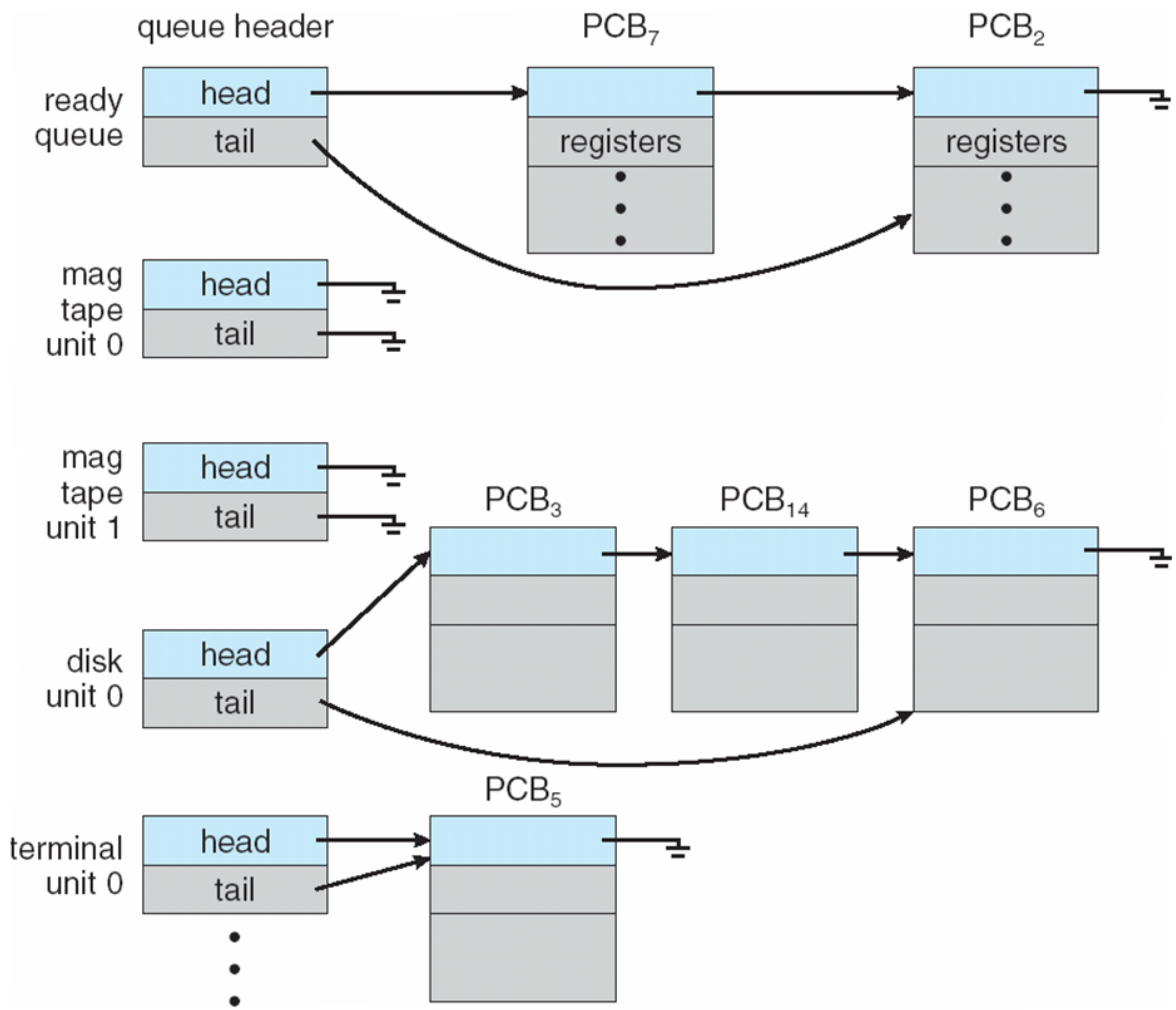
Process Scheduling

- 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



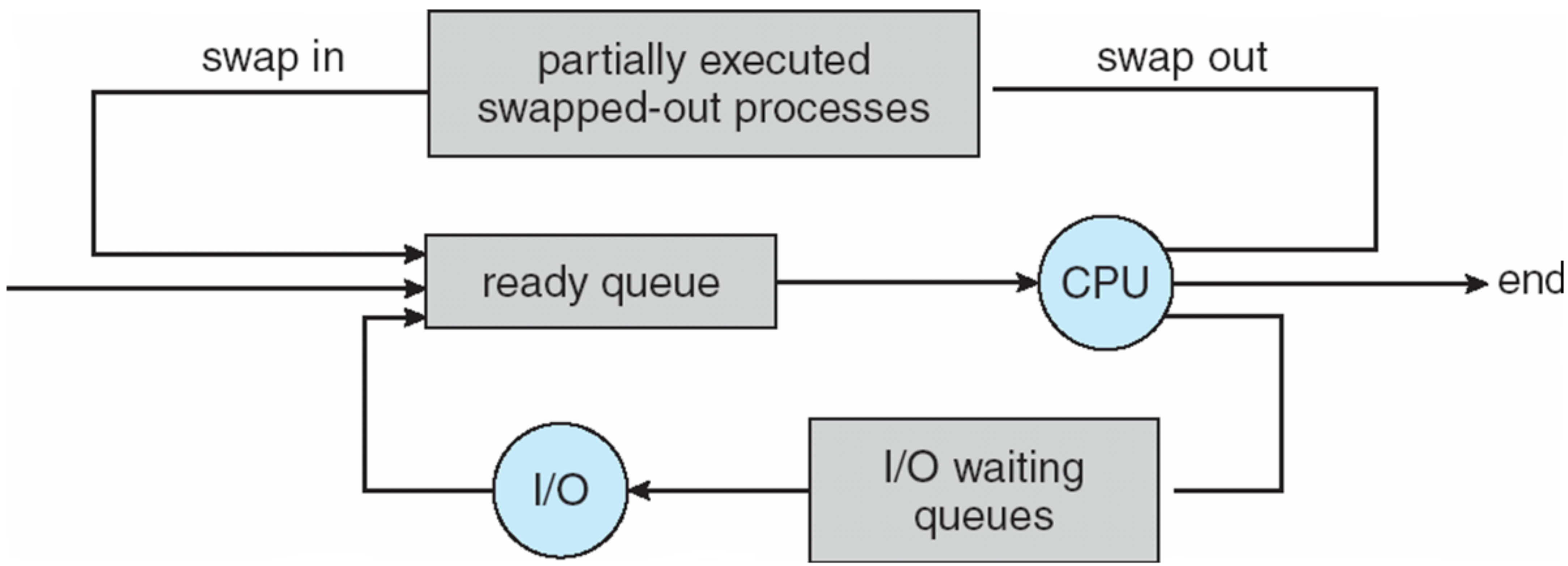
Ready Queue And Device Queues



Schedulers

- **Long-term scheduler** (or **job scheduler**)
  - **selects which processes should be brought into the ready queue**
  - long-term scheduler is invoked very infrequently
    - usually in seconds or minutes: it may be slow
  - long-term scheduler controls the degree of **multiprogramming**
- **Short-term scheduler** (or **CPU scheduler**)
  - **selects which process should be executed next and allocates CPU**
  - short-term scheduler is invoked very frequently
    - usually in milliseconds: it must be fast
  - sometimes the only scheduler in a system
- **Mid-term scheduler**
  - swap in/out partially executed process to relieve memory pressure

Medium Term Scheduling



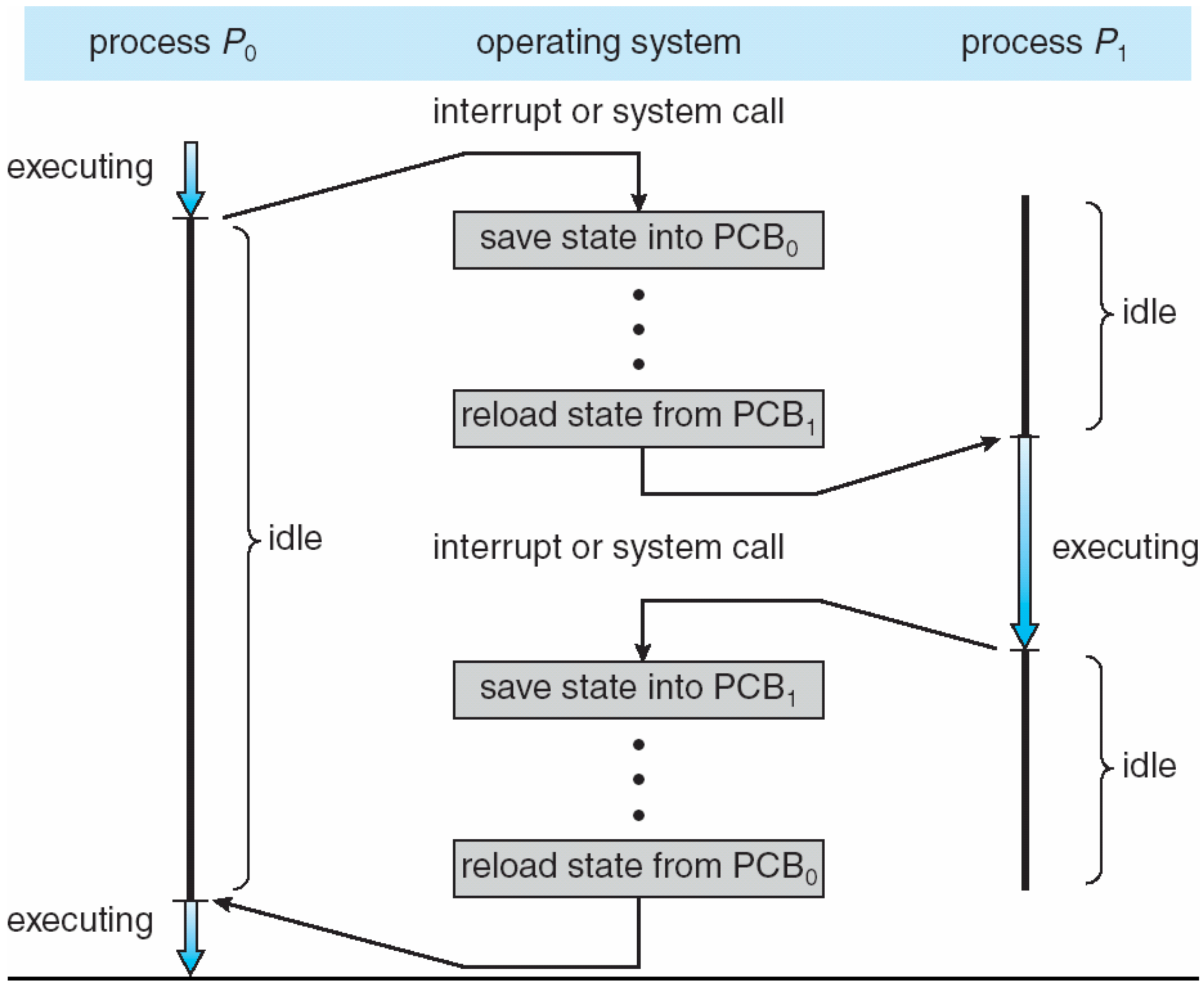
Scheduler

- 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



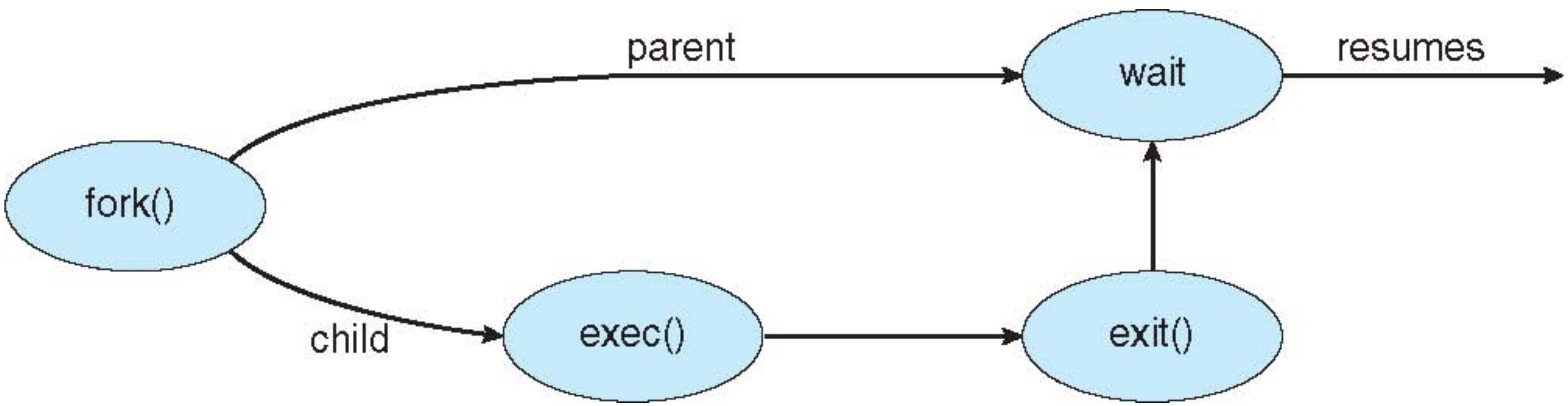
Process Creation

- 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

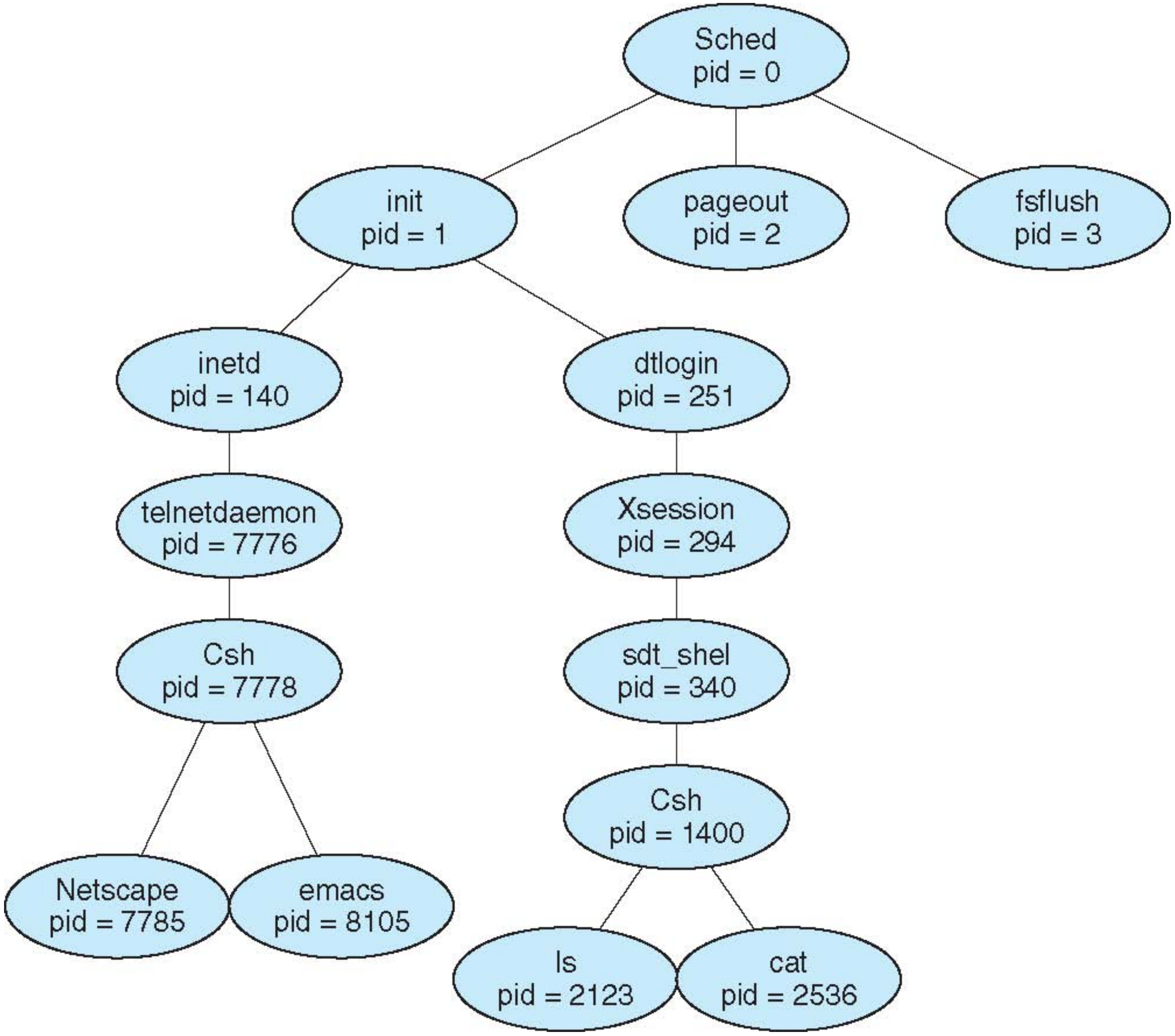
Process Creation



C Program Forking Separate Process

```c
#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;
}
```

A Tree of Processes on Solaris



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**

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)
- A common paradigm: **producer-consumer problem**
  - Producer process produces information that is consumed by a consumer process

Producer-consumer Based on Ring Buffer

- Shared data

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

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

Producer

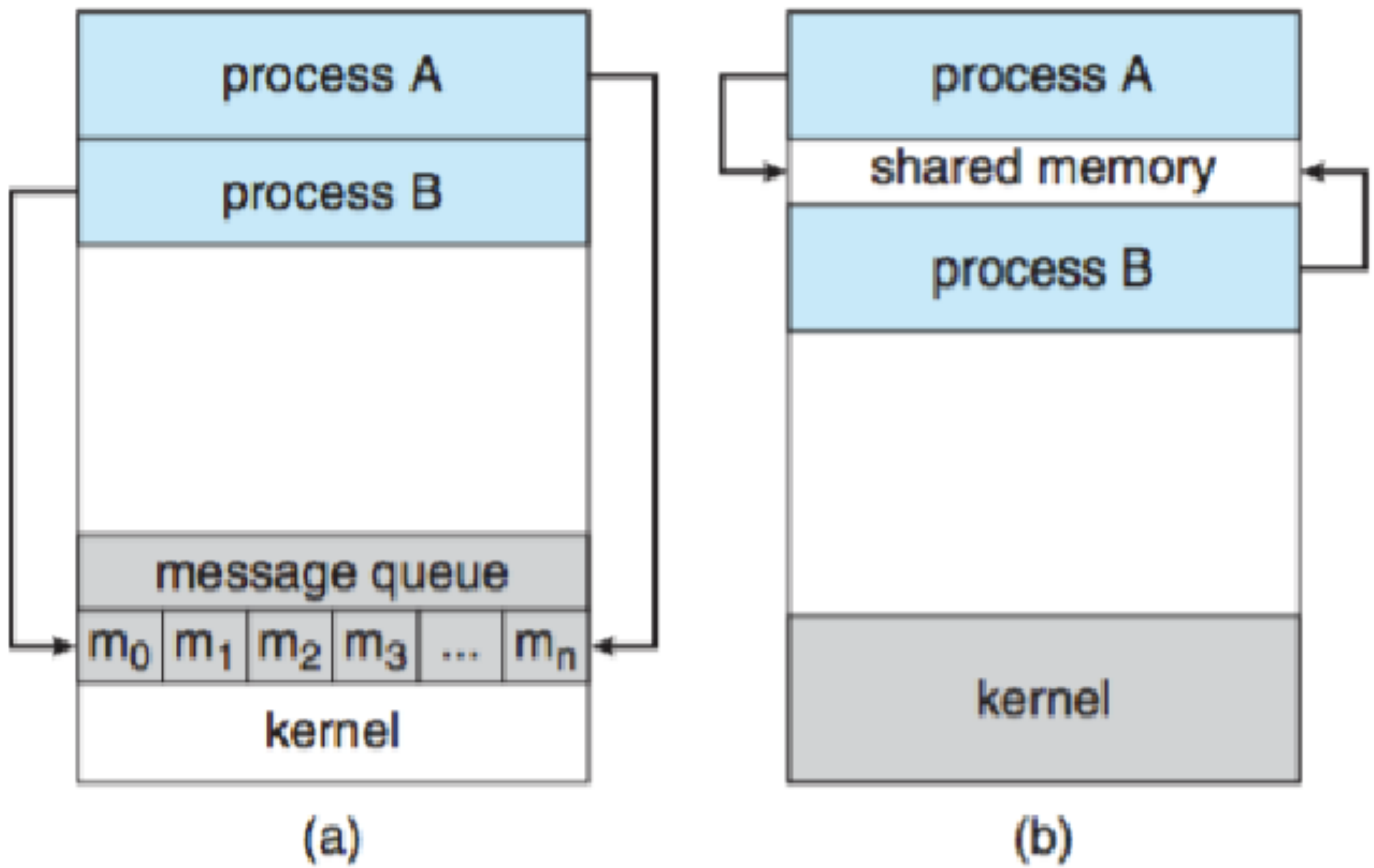
```
item nextProduced;
while (true) {
    /* produce an item in nextProduced*/
    while (((in + 1) % BUFFER_SIZE) == out)
        ;   /* do nothing -- no free buffers */

   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
  - how to utilize all the buffers? (job interview question)
    - without using one more variables?
  - need to synchronize access to buffer

Two Communication Models



Message Passing

Shared Memory

Shared Memory

- **Kernel** maps the same physical memory into the collaborating processes
  - might be at different virtual addresses
- Each process can access the shared memory independently & simultaneously
  - Access to shared memory must be **synchronized** (e.g., using **locks**)
- Shared memory is ideal for exchanging large amount of data

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 or pid-based
  - **exchange messages via send/receive**

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

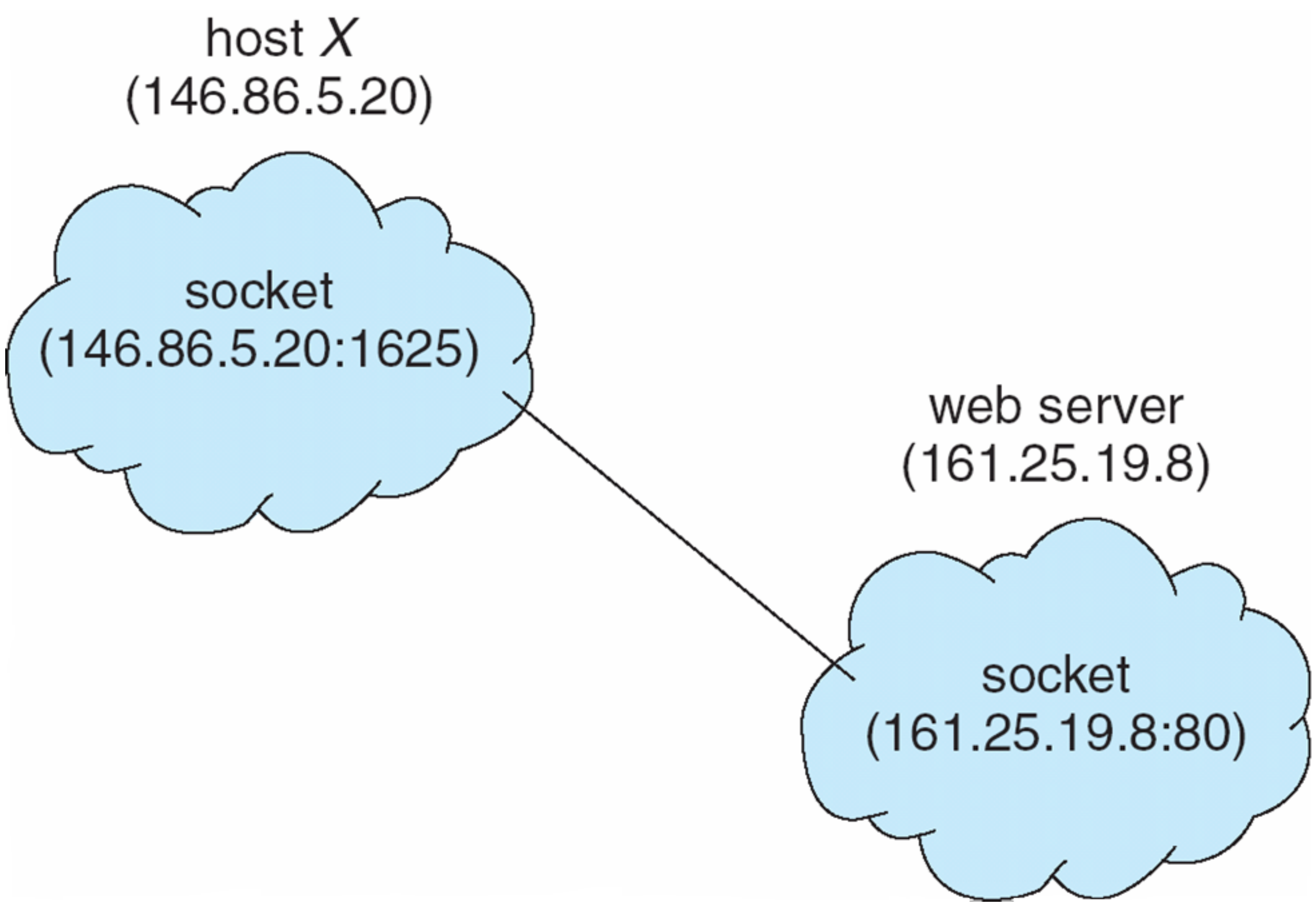
Example Message Passing Primitives

- Sockets
- Remote procedure calls
- Pipes
- 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**

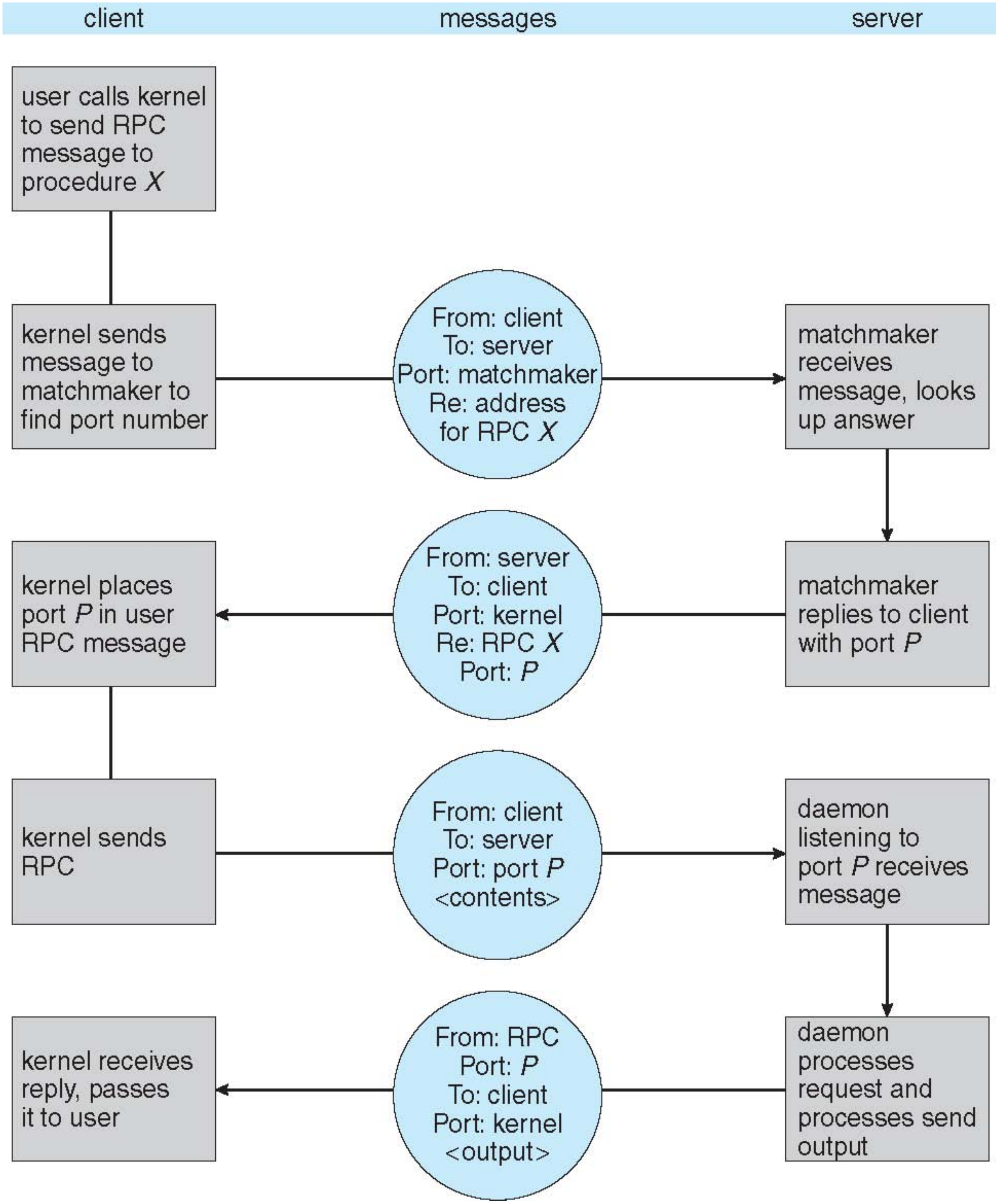
Socket Communication



Remote Procedure Call

- Remote procedure call (RPC) abstracts function calls between processes across networks
- **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

Execution of RPC



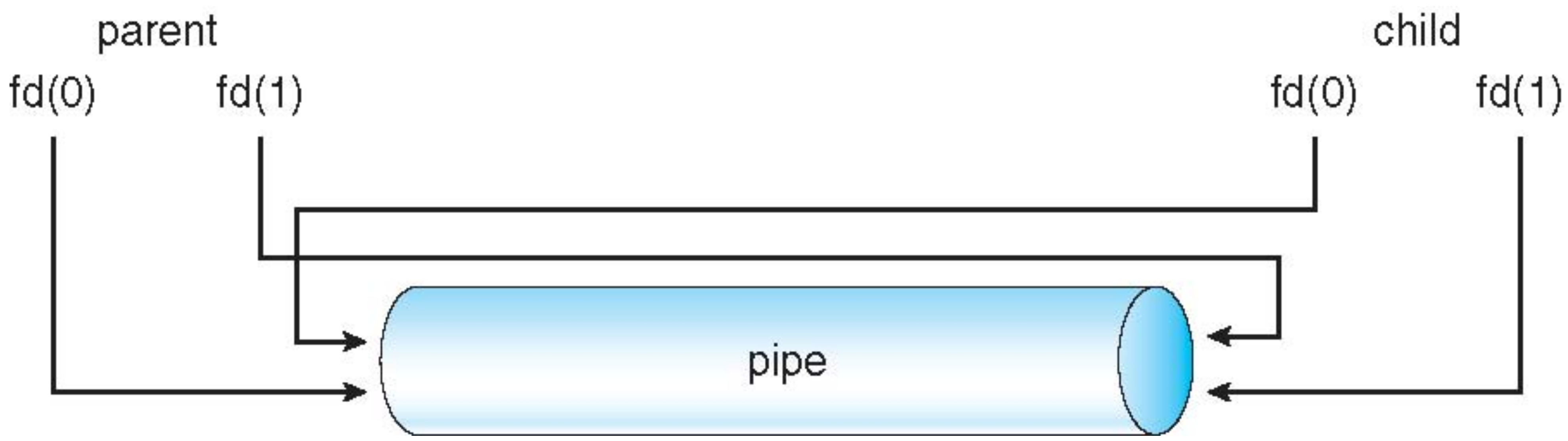
Pipes

- **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**
- Require **parent-child relationship** between communicating processes
- Activity: review Linux **man pipe**

Ordinary Pipes



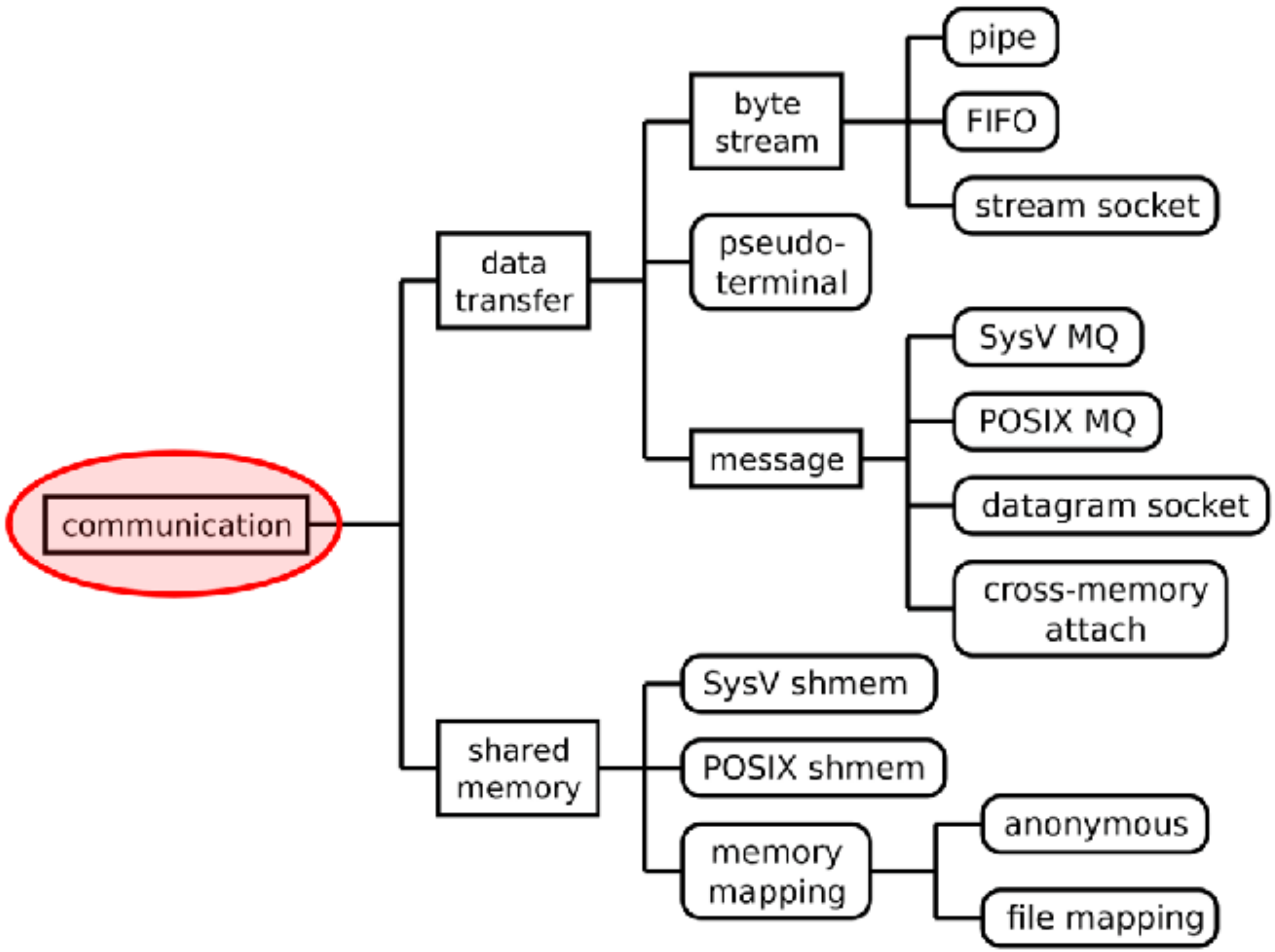
Named Pipes

- 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

Examples: Linux IPC

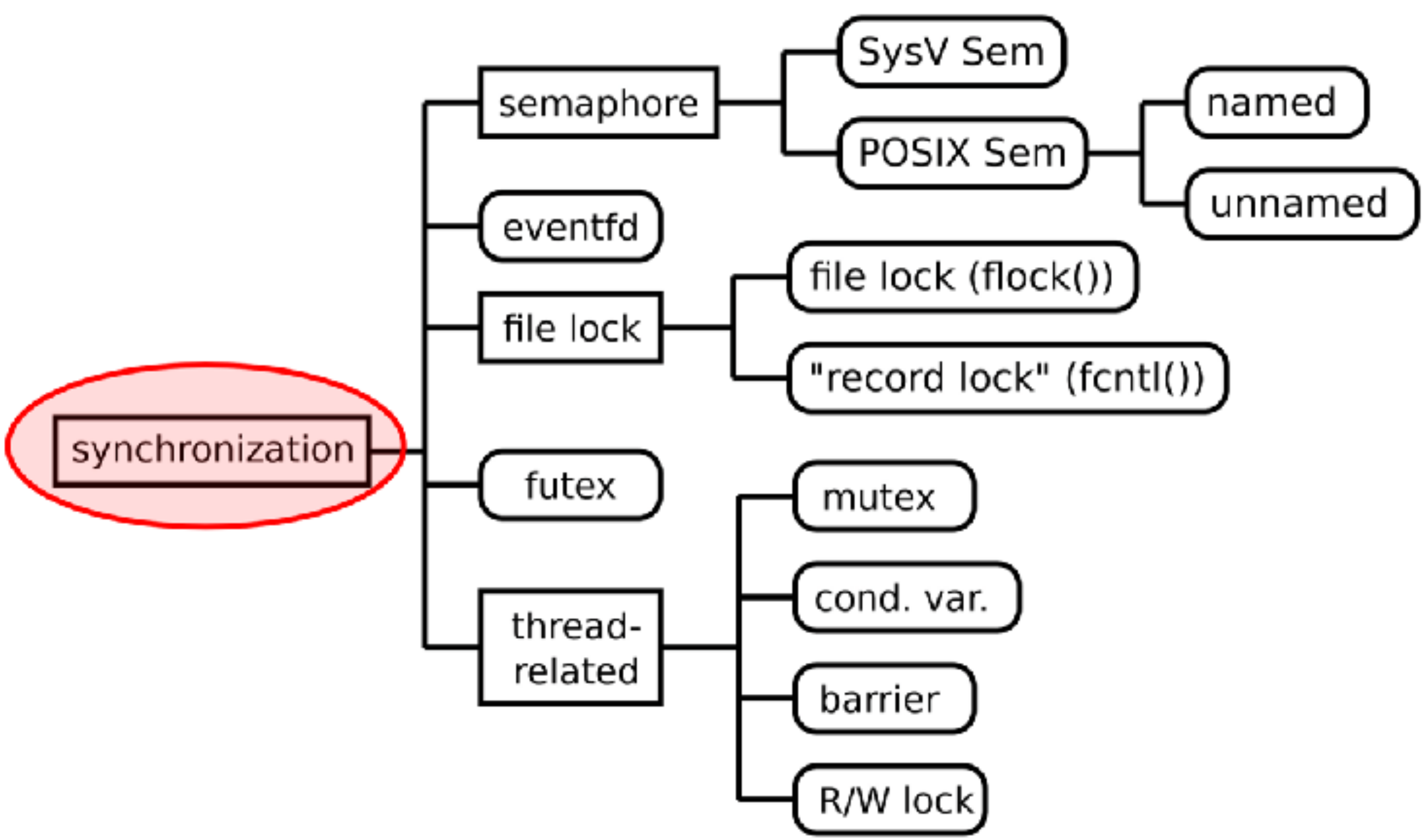
- **Communication:**
  - Pipes
  - Sockets
  - Shared memory
  - Message queues
  - Semaphores
  - …
- **Signals**
- **Synchronization**
  - Eventfd
  - Futexes
  - Locks
  - Condition variables
  - …

Linux IPC - Communication



source: http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf

Linux IPC - Synchronization



source: http://man7.org/conf/lca2013/IPC_Overview-LCA-2013-printable.pdf

Linux IPC: System V Shared Memory

- Process first creates shared memory segment

    segment id = **shmget**(key, size, flag);

- Process wanting access to that shared memory must attach to it

    shared memory = (char *) **shmat**(id, NULL, 0);

- Now the process could write to the shared memory
- When done, a process can detach the shared memory

    shmdt(shared memory);

End of Chapter 3