PROCESSES

Ruben Acuña

Spring 2018

Kernel source code is coming from Linux 4.9.6. See https://www.kernel.org/

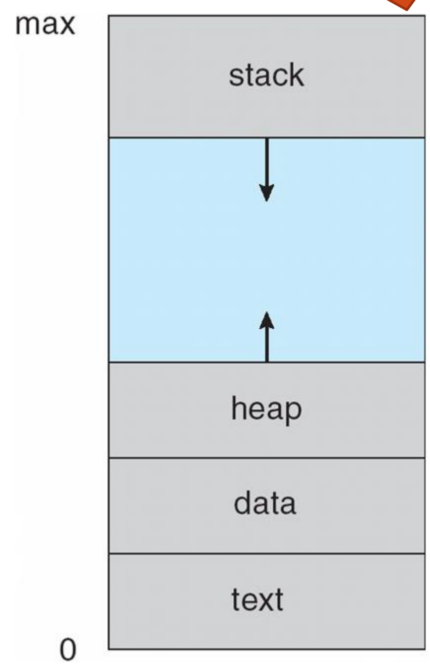
2

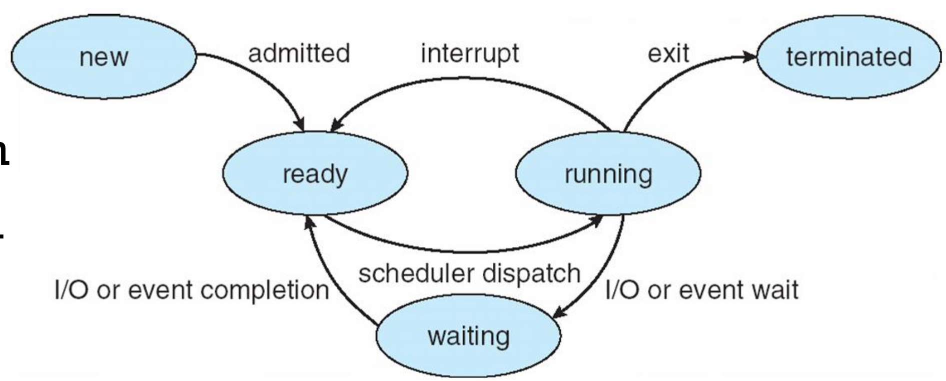
PROCESS CONCEPT (.1)

We only have one $sp register – how do we support multiple processes?

PROCESS AND STATE

- Roughly, a process is to an executable what an object is to a class.
  - Book also mentions the term *job*…
- An executing process has storage for stack, heap, "data", and "text".
  - Really: picture shows a "virtualized environment" that program is supposed to execute within.
- A general abstraction for the state of a process is shown to the right. In general, most operating systems will have these states although they may be named differently, or some may be combined/split. (Linux example to come.)





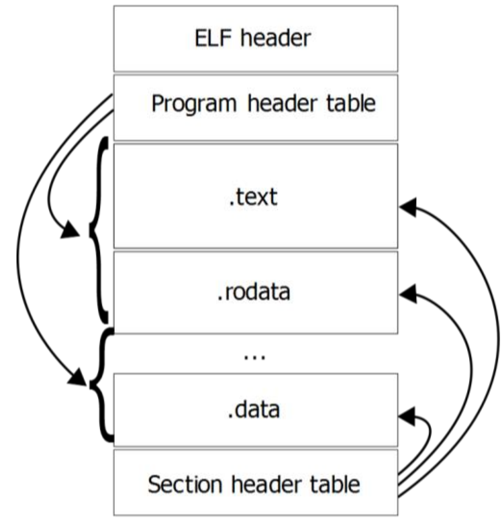
Images from Operating System Concepts.

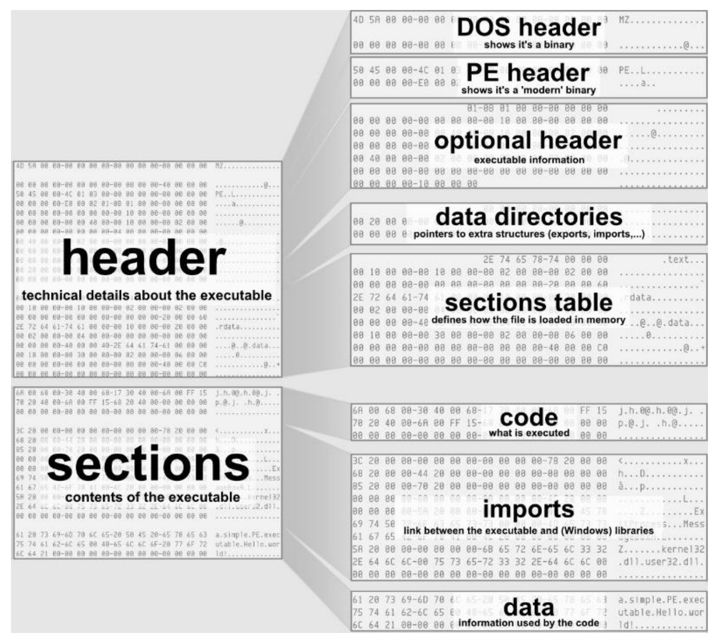
EXECUTABLE FORMATS

- All executables are obligated to follow a specific *Application Binary Interface* (ABI) that specifies system properties like type size, and library interface mechanism.

- Executable and Linkable Format (ELF; Unix)
  - Includes header, program, and various other sections (data).

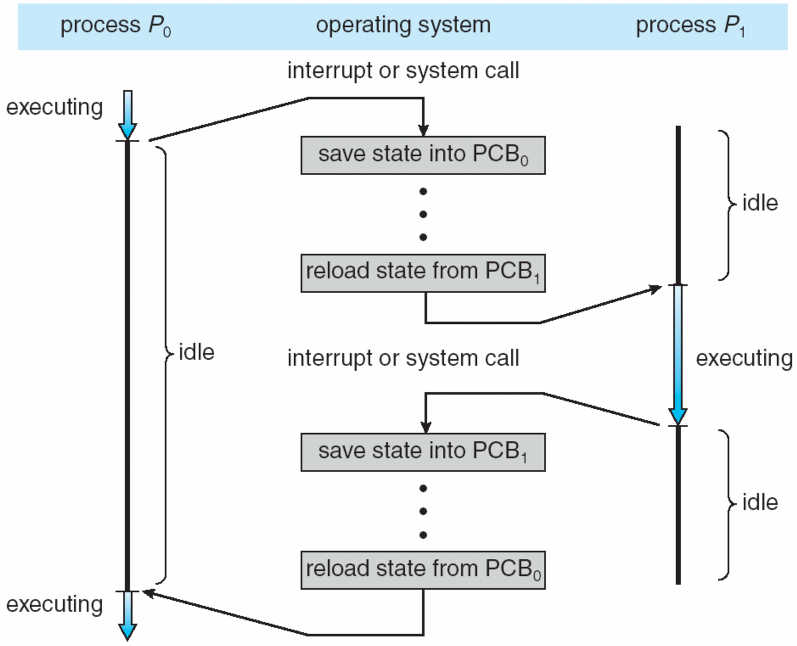
- Portable Executable (PE; Win32)
  - Includes legacy MS DOS header, followed by modern PE header, and associated sections (text, data, rdata).





Images from Wikipedia and corkami.com.

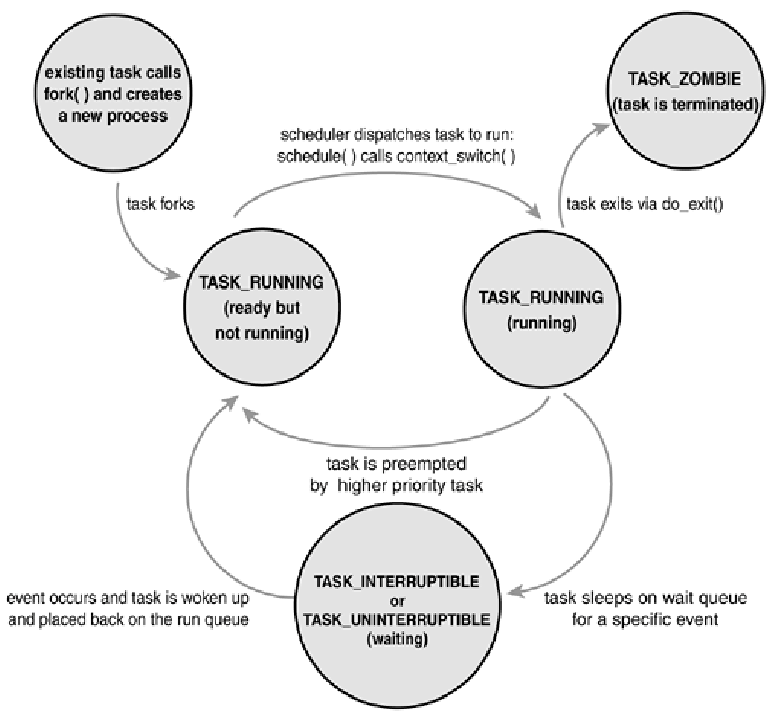
PROCESS CONTROL BLOCK



- Recall that processes live in a "virtualized environment" – they act as if they the only thing running on the CPU.
- We also need the ability to start and stop processes at will.
- Ergo, the entire state of a process is captured with something called an Process Control Block (PCB), which contains:
  - Process state
  - Program counter
  - CPU registers
  - CPU-scheduling information
  - Memory management information
  - Accounting information
  - I/O status information

STRUCT TASK_STRUCT;

- In Linux, processes are represented with a *process descriptor* struct.
- Process descriptors are stored in a circular doubly linked list.




```
struct task_struct {
   volatile long state;        /* -1 unrunnable, 0 runnable, >0 stopped *
   void *stack;
   unsigned int flags;         /* per process flags, defined below */
   unsigned int ptrace;

   int prio, static_prio, normal_prio;
   unsigned int rt_priority;
   const struct sched_class *sched_class;

   unsigned int policy;
   int nr_cpus_allowed;
   cpumask_t cpus_allowed;

  /* task state */
   int exit_state;
   int exit_code, exit_signal;
   int pdeath_signal;  /*  The signal sent when the parent dies  */
   unsigned long jobctl;       /* JOBCTL_*, siglock protected */

   pid_t pid;
   pid_t tgid;

   struct task_struct __rcu *real_parent; /* real parent process */
   struct task_struct __rcu *parent; /* recipient SIGCHLD, wait4() report
   struct list_head children;   /* list of my children */
   struct list_head sibling;    /* linkage in my parent's children list */
   struct task_struct *group_leader;         /* threadgroup leader */

    /* RA: ~400 lines omitted. Mostly preprocessor directives. */

   struct thread_struct thread;
};
```


Images from makelinux.net.

TASK_STRUCT MEMORY ALLOCATION

- task_structs come and go quickly, but memory management is slow - how to fix?

8

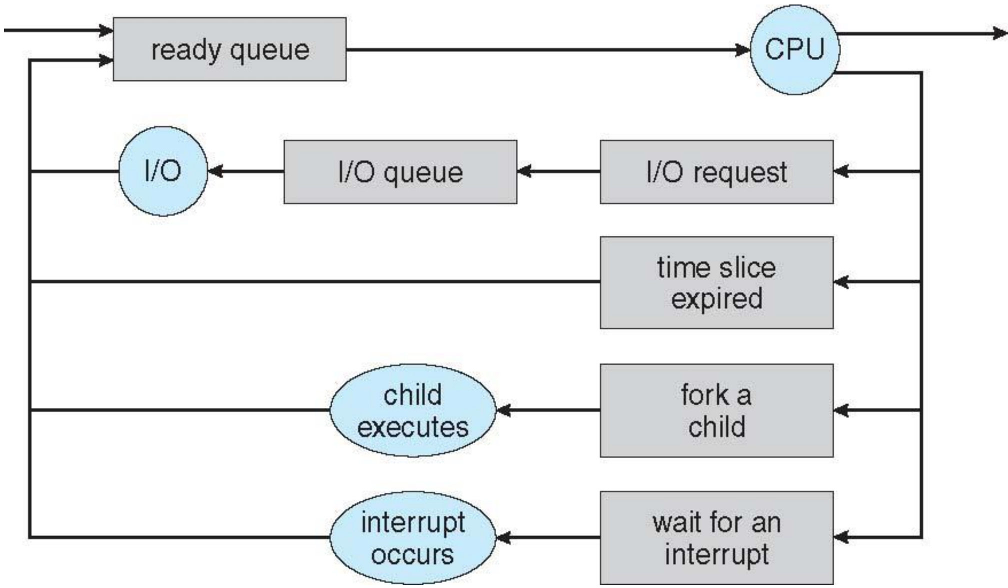
PROCESS SCHEDULING (.2)

SCHEDULING QUEUES

- Given that we want to support executing more than one process at a time, OSs need some sort of *process scheduler* that divides up resources (CPU time).

- We have two general notations:
  - *Ready queue*: processes ready to execute on the CPU.
  - *Device queue*: processes waiting to use a device/resource.

- Whenever a process from the ready queue is given CPU time, we say that it is *dispatched*.



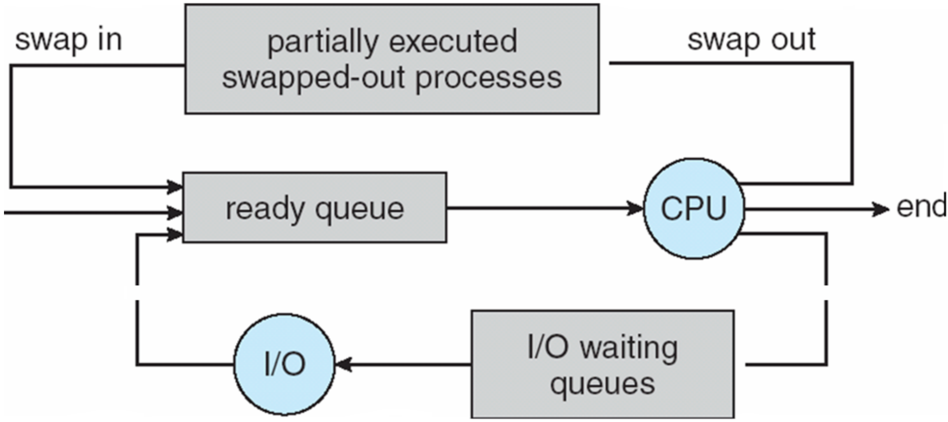
Queueing Diagram – rectangles indicates queues, ovals indicate resources.

Would it make sense to use priority queues?

Image from Operating System Concepts.

SCHEDULERS

- There 2 patterns of process resource utilization:
  - IO-bound
  - CPU-bound
  - If we want to maximum performance, what ratio of these should we have and why?



Adding *swapping* into queueing system.

- There are 3 types of schedulers:
  - Short-term scheduler (RAM)
  - Medium-term scheduler (swap)
  - Long-term scheduler (Disk)

- During process execution, we say that the processes‘ *context* is active.
- If we want to change the running process, then we must preform a *context switch* to *restore* a PCB's state. We must do a state *save* as well.

Image from Operating System Concepts.

11

OPERATIONS ON PROCESSES (.3)

CREATION

- As seen in task_struct, each process is identified by a pid.
  - In general, pid2 > pid1 which indiates that pid2 was created later.

- The first process to be executed is called init and has pid=1. Since each process contains the pid of it's parent, we can view the relation between processes as a tree rooted at init.
- We have some things to think about:
  - Where should the children's resources come from?
  - What should the parent do after creating the child?
  - What code should the child use?

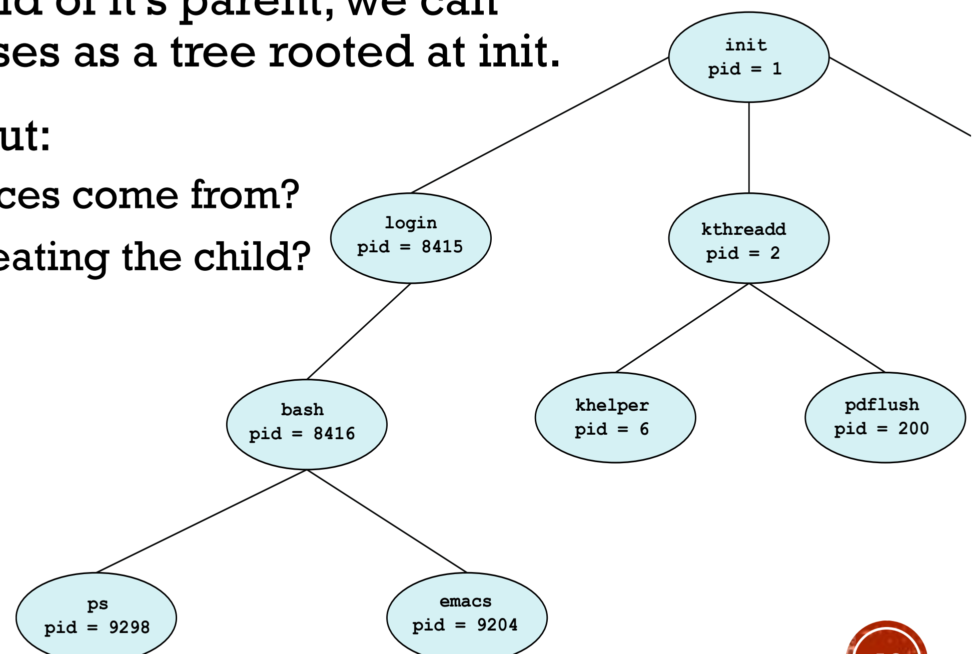


Image from Operating System Concepts.

POSIX Sample – Splitting Execution

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

int main() {
        pid_t pid;
        pid = fork();

        if (pid < 0) {
                //What process does this execute in?
                fprintf(stderr, "Fork Failed");
                return 1;
        }
        else if (pid == 0) {
                //What process does this execute in?
                execlp("/bin/ls", "ls", NULL);
        }
        else {
                //What process does this execute in?
                wait(NULL);
                printf("Child Complete");
        }
        return 0;
}
```

Under POSIX, fork() results in a child process that take is copied from the parent. Does copying seem like a good idea?

Code modified from Operating System Concepts.

For WinAPI coding conventions, see https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx

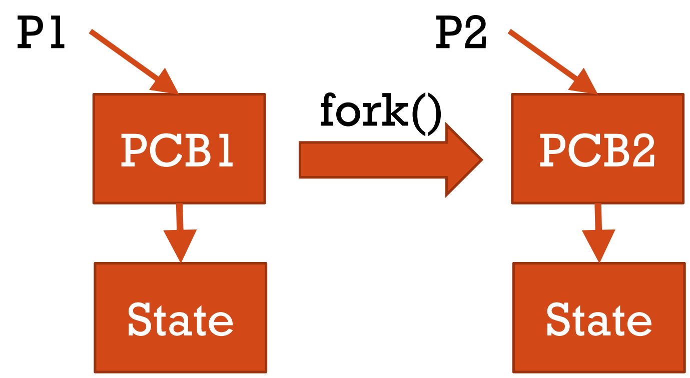
WINAPI - EXECUTING COMMAND

```
#include <stdio.h>
#include <windows.h>
int main(VOID) {
          STARTUPINFO si;
          PROCESS_INFORMATION pi;
          ZeroMemory(&si, sizeof(si));              /* RA: DOES NOT allocate memory */
          si.cb = sizeof(si);
          ZeroMemory(&pi, sizeof(pi));
          if (!CreateProcess(NULL,                  /* lpApplicationName */
              "C:\\WINDOWS\\system32\\mspaint.exe", /* lpCommandLine */
              NULL,                                 /* lpProcessAttributes */
              NULL,                                 /* lpThreadAttributes */
              FALSE,                                /* bInheritHandles */
              0,                                    /* dwCreationFlags */
              NULL,                                 /* lpEnvironment */
              NULL,                                 /* lpCurrentDirectory */
              &si,
              &pi)) {
                  fprintf(stderr, "Create Process Failed");
                  return -1;
          }
          WaitForSingleObject(pi.hProcess, INFINITE);
          printf("Child Complete");
          CloseHandle(pi.hProcess);
          CloseHandle(pi.hThread);
}
```

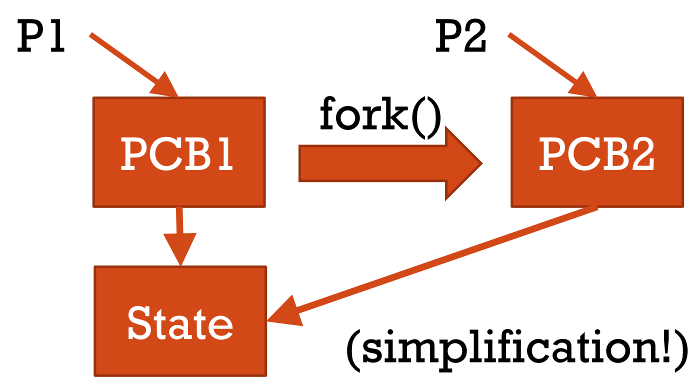

Code modified from Operating System Concepts.

IMPLEMENTING FORK()

- Here's a thought: isn't it expensive for fork() to create a copy of the original process if we immediately exec() over it?



- So, why not just omit copying all of the process state and instead use a reference pointer to it? Only when the execution of the forked process requires a write, then, perform the copy.
- This technique is called *copy-on-write* and can seen as *lazy* method of implementing *deep clone* operation on a data structure.



TERMINATION

- To terminate a process, one invokes exit(n);
  - This is the implicit result of the return statement in main.
  - Or, may be invoked by an another program (parent, system, user, etc.).
- What are the reasons a parent might terminate a child?
- If a parent terminates, then typically each descendent process is also terminated (*cascading termination*). (Is this a good idea?)
- Under POSIX, we can use wait(&n) to "wait" for a child process to terminate. (Means we must save the pid from fork.)
- If a process terminates before it's parent has called wait(), then it cannot fully be removed from the process tree. (Why?) It is a called a *zombie* process. If somehow the parent is lost, then the executing child process may become *orphan*ed.

17

INTERPROCESS COMMUNICATION (.4)

COMMUNICATION TYPES

- The process model we have seen so far is for *independent* processes, i.e., those which do not share data. The alternative is *cooperative* processes. There are reasons for cooperation:
  - Information Sharing
  - Computation Speedup
  - Modularity
  - Convenience

- Recall earlier, the two mechanisms of IPC we discussed: Shared Memory and Message-Passing.
  - Are there times when one in particular is necessary?
  - Which is faster in terms of system calls?

Note that this is all conceptual! The APIs are different.

SHARED-MEMORY SYSTEMS

- Producer-Consumer Problem:
  - Some process P1 creates products that are to be consumed by process P2.
- Let's try solving this problem with a buffered shared-memory solution.
- Does this code seem safe?

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

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


//Producer
item next_produced;
while (true) {
    /* produce next_produced item */
    while (((in + 1) % BUFFER_SIZE) == out);
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

//Consumer
item next_consumed;
while (true) {
           while (in == out);
           next_consumed = buffer[out];
           out = (out + 1) % BUFFER_SIZE;
           /* consume next_consumed item*/
}
```


Code modified from Operating System Concepts.

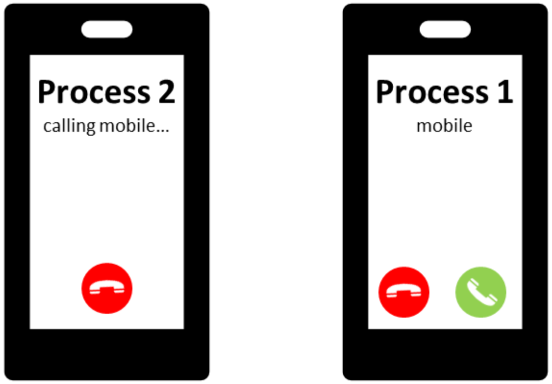
MESSAGE-PASSING SYSTEMS

- At minimum, we need functions send(msg) and receive(msg).
  - How much does it matter if the message is fixed length or variable length?
- If processes P1 and P2 want to communicate, we must establish a *communication link*.
- Processes may use *direct communication*. Links are complete, pairwise, and unique.
  - Symmetric: send(P1, message), receive(P2, message)
  - Asymmetric: send(P, message), receive(&id, message)
- Processes may also use *indirect communication* over a *mailbox/port*. Links are specific, groupwise, and non-unique.
  - send(M, message), receive(M, message)
  - Consider the following problem: P1 sends a message to port K, but then P2 and P3 do a receive on K. What should happen? Is there an issue to fix?

MESSAGE-PASSING SYSTEMS

- For messaging commands we have a choice of either waiting for an answer (Blocking/Synchronous), or just continuing on (Nonblocking/Asynchronous):
  - Blocking Send
  - Non-blocking Send
  - Blocking Receive
  - Non-blocking receive
- By picking the correct operations, can we easily "solve" the producer-consumer problem…?
- There are three ways we can design a buffer:
  - Zero Capacity
  - Bounded Capacity
  - Unbounded Capacity

Blocking



Non-blocking

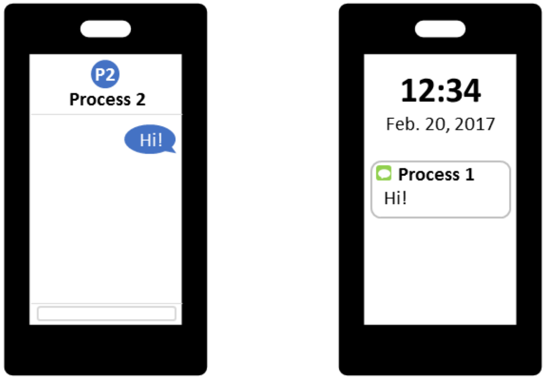


Image by Marcinina Alvaran (Spring B 2017, UGTA).

MESSAGE-PASSING SYSTEMS

- For messaging commands we have a choice of either waiting for an answer (Blocking/Synchronous), or just continuing on (Nonblocking/Asynchronous):
  - Blocking Send
  - Non-blocking Send
  - Blocking Receive
  - Non-blocking receive
- By picking the correct operations, can we easily "solve" the producer-consumer problem…?
- There are three ways we can design a buffer:
  - Zero Capacity
  - Bounded Capacity
  - Unbounded Capacity

```
//producer
while (true) {
        /* produce next_produced item */
        send(next_produced);
}
//consumer
while (true) {
        receive(next_consumed);

        /* consume next_consumed item */
}
```

23

EXAMPLES OF IPC SYSTEMS (.5)

POSIX – SHARED MEMORY

- First, let's look at shared memory in POSIX. It functions via the idea of a memory-mapped file that one process creates and the other accesses.
- The next slide has a full program but four functions in particular are key:
  - shm_open()
  - ftruncate()
  - mmap()
  - shm_unlink()

PRODUCER

```
#include <stdio.h>
#include <stlib.h>
#include <string.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    const char *message_0 = "Hello";
    const char *message_1 = "World!";

    int shm_fd;

    void *ptr;

    shm_fd = shm_open(name, O_CREAT | O_RDRW, 0666);        //shm_open usage

    ftruncate(shm_fd, SIZE);                                //ftruncate usage

    ptr = mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0); //mmap usage

    sprintf(ptr, "%s", message_0);
    ptr += strlen(message_0);
    sprintf(ptr, "%s", message_1);
    ptr += strlen(message_1);
    return 0;
}
```


Code modified from Operating System Concepts.

CONSUMER

```c
#include <stdio.h>
#include <stlib.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/stat.h>
int main() {
    const int SIZE = 4096;
    const char *name = "OS";
    int shm_fd;
    void *ptr;

    shm_fd = shm_open(name, O_RDONLY, 0666);               //shm_open usage

    ptr = mmap(0, SIZE, PROT_READ, MAP_SHARED, shm_fd, 0); //mmap usage

    printf("%s", (char *)ptr);

    shm_unlink(name);                                      //shm_unlink usage
    return 0;
}
```


Code modified from Operating System Concepts.

MACH – MESSAGE PASSING

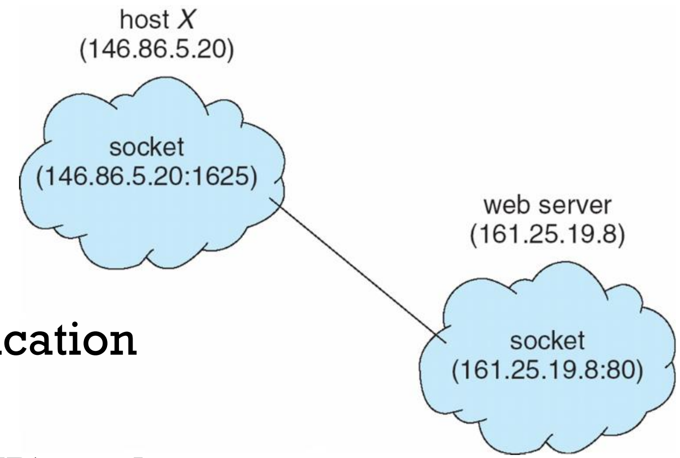
- Processes communicate via *messages* passed through *ports (buffers)*.
- API includes:
  - msg_send()
  - msg_receive()
  - msg_rpc()
  - port_allocate()
    - Messages are stored FIFO but per sender.
- If ports are named, they can support n-1, 1-n, n-n communication. (…but not *broadcasts*…)

header			data		
src	dst	size	var1	var2	var3

28

COMMUNICATION IN CLIENT-SERVER SYSTEMS (.6)

SOCKETS



- One concept for the endpoint of a communication channel between systems is a socket.
- A socket is composed of a system's id (i.e., IP) and a port number. Ex: 98.175.148.150:80.
  - Specific ports below 1024 have semantics attached, e.g., 80 is HTTP.
- On the next slide, we'll take a look at establishing sockets and using them for communication in Java. (C is much more messy.)
  - The example will use the TCP protocol, but is another common network protocol called UDP.

USING SOCKETS IN JAVA

```
import java.net.*;
import java.io.*;
public class DateServer {
  public static void main(String[] args) {
    try {
      ServerSocket sock = new ServerSocket(6013);

      while (true) {
        Socket client = sock.accept();
        PrintWriter pout = new
          PrintWriter(client.getOutputStream(), true);

        pout.println(new java.util.Date().toString());

        client.close();
      }
    }
    catch (IOException ioe) {
        System.err.println(ioe);
    }
  }
}
```

```
import java.net.*;
import java.io.*;
public class DateClient {
  public static void main(String[] args) {
    try {
      Socket sock = new Socket("127.0.0.1",6013);
      InputStream in = sock.getInputStream();
      BufferedReader bin = new
        BufferedReader(new InputStreamReader(in));

      String line;
      while ( (line = bin.readLine()) != null)
        System.out.println(line);

      sock.close();
    }
    catch (IOException ioe) {
        System.err.println(ioe);
    }
  }
}
```

"localhost"

Source code from Operating System Concepts.

REMOTE PROCEDURE CALLS

- The sockets example we just did is low level – we moved a stream of characters (bytes) across processes.
- A higher level mechanism is Remote Procedure Calls (RPCs).
  - Sends a well-structured packet which contains procedure identifier and input.
- Provided API *stubs* exist to *marshal* parameters into RPC format, clients act as if they are simply calling a function.
- Since calls may have some impact on the target system (i.e., they don't just return a value), we need to be sure they execute exactly once.
  - How?
- Also: RPCs are linked to specific ports so we need something like a *matchmaker* to look up the server's port.

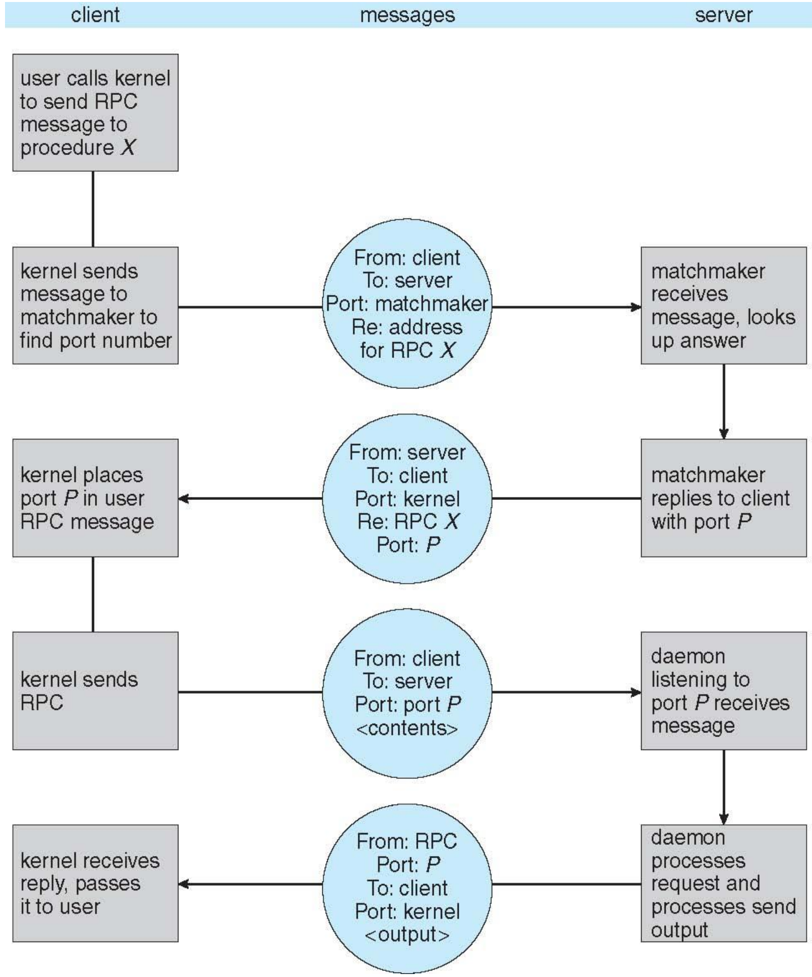
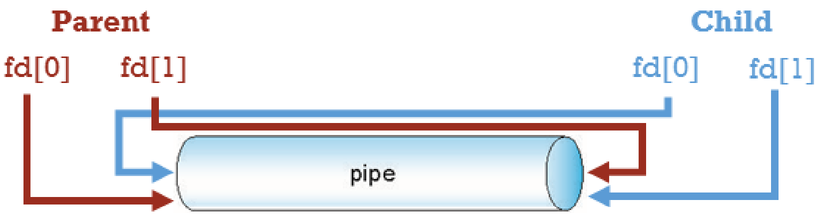


Image from Operating System Concepts.

PIPES

- A pipe is a UNIX concept for passing information between processes.
- Some considerations:
  - Bidirectional or unidirectional?
  - Half duplex or full duplex?
  - Are processes asymmetric?
  - Networked or local?



- There are two types of pipes:
  - *Ordinary*: only created in the context of a single process.
    - Have *read-end* and *write-end*. No seek ability.
  - *Named*: explicitly named resource that can be accessed by multiple processes and is persistent. Similar to a file.

Image adapted from Operating System Concepts.

PIPES

```
#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>
#define BUFFER_SIZE 25
#define READ_END 0
#define WRITE_END 1
int main(void) {
  char write_msg[BUFFER_SIZE] = "Greetings";
  char read_msg[BUFFER_SIZE];
  int fd[2];
  pid_t pid;

  if (pipe(fd) == -1) {
    fprintf(stderr, "Pipe failed");
    return 1;
  }
  pid = fork();
  if (pid > 0) { /* parent process */
    close(fd[READ_END]);
    write(fd[WRITE_END], write_msg, strlen(write_msg) + 1);
    close(fd[WRITE_END]);
  }
  else { /* child process */
    close(fd[WRITE_END]);
    read(fd[READ_END], read_msg, BUFFER_SIZE);
    printf("read %s", read_msg);
    close(fd[READ_END]);
  }
  return 0;
}
```