Processes

CS201 Lecture 3

Jason Hibbeler

University of Vermont

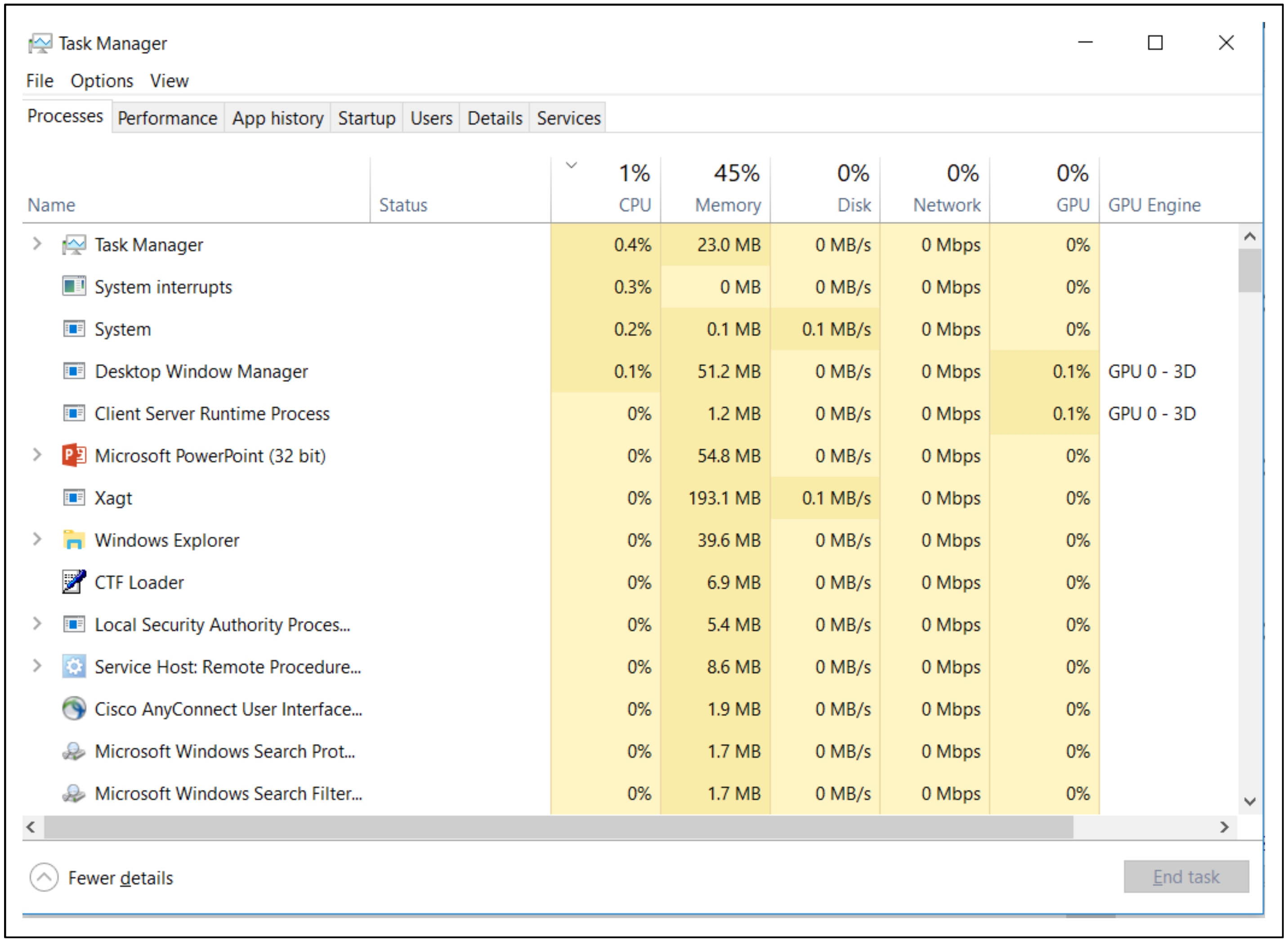
Fall 2019

A Process

Process: thing that’s running on a computer

- a process is a program that is in execution
- could be a user’s job
- could be a task from the OS itself
- note that even in a single-user environment (such as on my PC), there are many processes running at any given time

My PC: Snapshot of Processes



Task Manager

File Options View

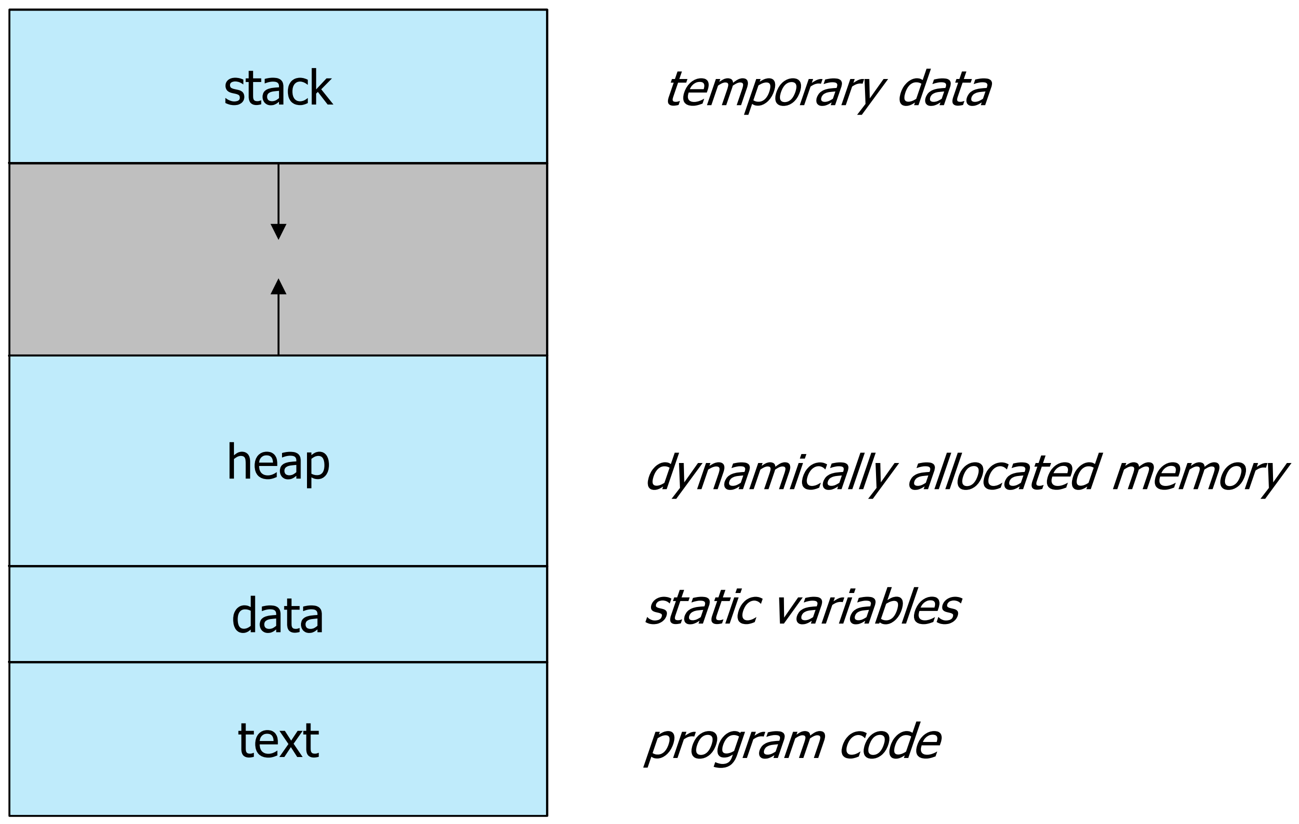
Processes | Performance | App history | Startup | Users | Details | Services

Name	Status	1% CPU	45% Memory	0% Disk	0% Network	0% GPU	GPU Engine
Task Manager		0.4%	23.0 MB	0 MB/s	0 Mbps	0%	
System interrupts		0.3%	0 MB	0 MB/s	0 Mbps	0%	
System		0.2%	0.1 MB	0.1 MB/s	0 Mbps	0%	
Desktop Window Manager		0.1%	51.2 MB	0 MB/s	0 Mbps	0.1%	GPU 0 - 3D
Client Server Runtime Process		0%	1.2 MB	0 MB/s	0 Mbps	0.1%	GPU 0 - 3D
Microsoft PowerPoint (32 bit)		0%	54.8 MB	0 MB/s	0 Mbps	0%	
Xagt		0%	193.1 MB	0.1 MB/s	0 Mbps	0%	
Windows Explorer		0%	39.6 MB	0 MB/s	0 Mbps	0%	
CTF Loader		0%	6.9 MB	0 MB/s	0 Mbps	0%	
Local Security Authority Proces...		0%	5.4 MB	0 MB/s	0 Mbps	0%	
Service Host: Remote Procedure...		0%	8.6 MB	0 MB/s	0 Mbps	0%	
Cisco AnyConnect User Interface...		0%	1.9 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Search Prot...		0%	1.7 MB	0 MB/s	0 Mbps	0%	
Microsoft Windows Search Filter...		0%	1.7 MB	0 MB/s	0 Mbps	0%	

Fewer details

End task

Parts of a Process

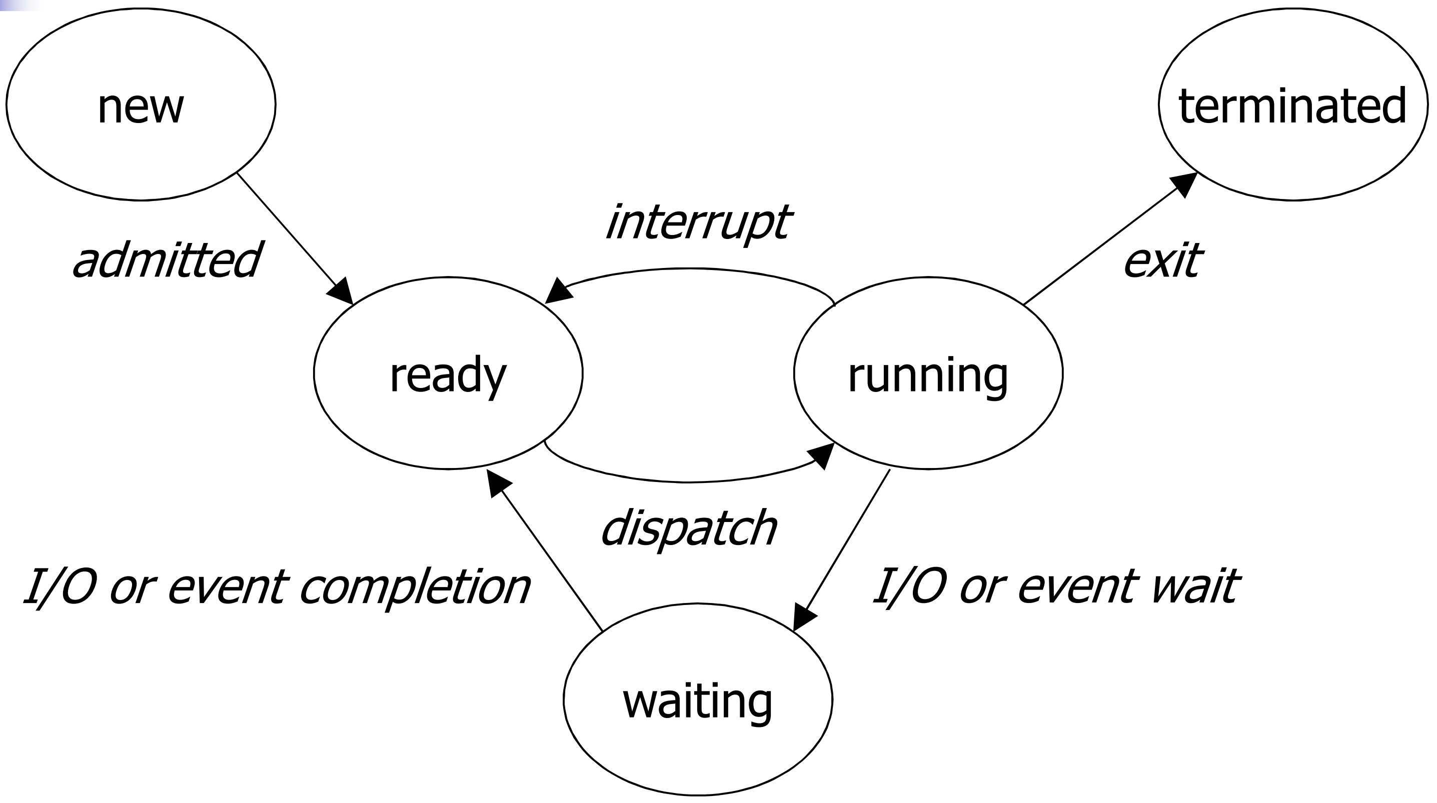


Parts of a Process

A process might be an environment for another program to execute

- a Java virtual machine
- an Android emulator

Process State



Key fact: only one process at a time can be running on a processor element at any given time; many processes can be ready or waiting

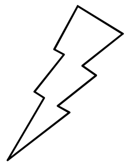
PCB: Process Control Block

The OS represents every process in the system using a PCB

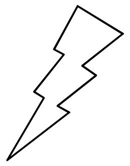
- contains the full state of the process at a point in time
- it's like a snapshot of the process at a specific point in time

General idea here is that the PCB has enough info so that the OS can interrupt and resume the process.

...running...running...running



process is suspended: save state to PCB for the process



running...running...

some time later, the process will resume: read state from the PCB of the process

PCB: Process Control Block

The OS represents every process in the system using a PCB

- process state
- program counter
- CPU registers
- CPU-scheduling information: to help the OS schedule the process
- memory-management information
- accounting information
- I/O status information: info about devices allocated to the process; open files

Thread: a Single Stream of Execution

A thread is like a scaled-down version of a process that lives within a process

- threads share resources among each other within the parent process

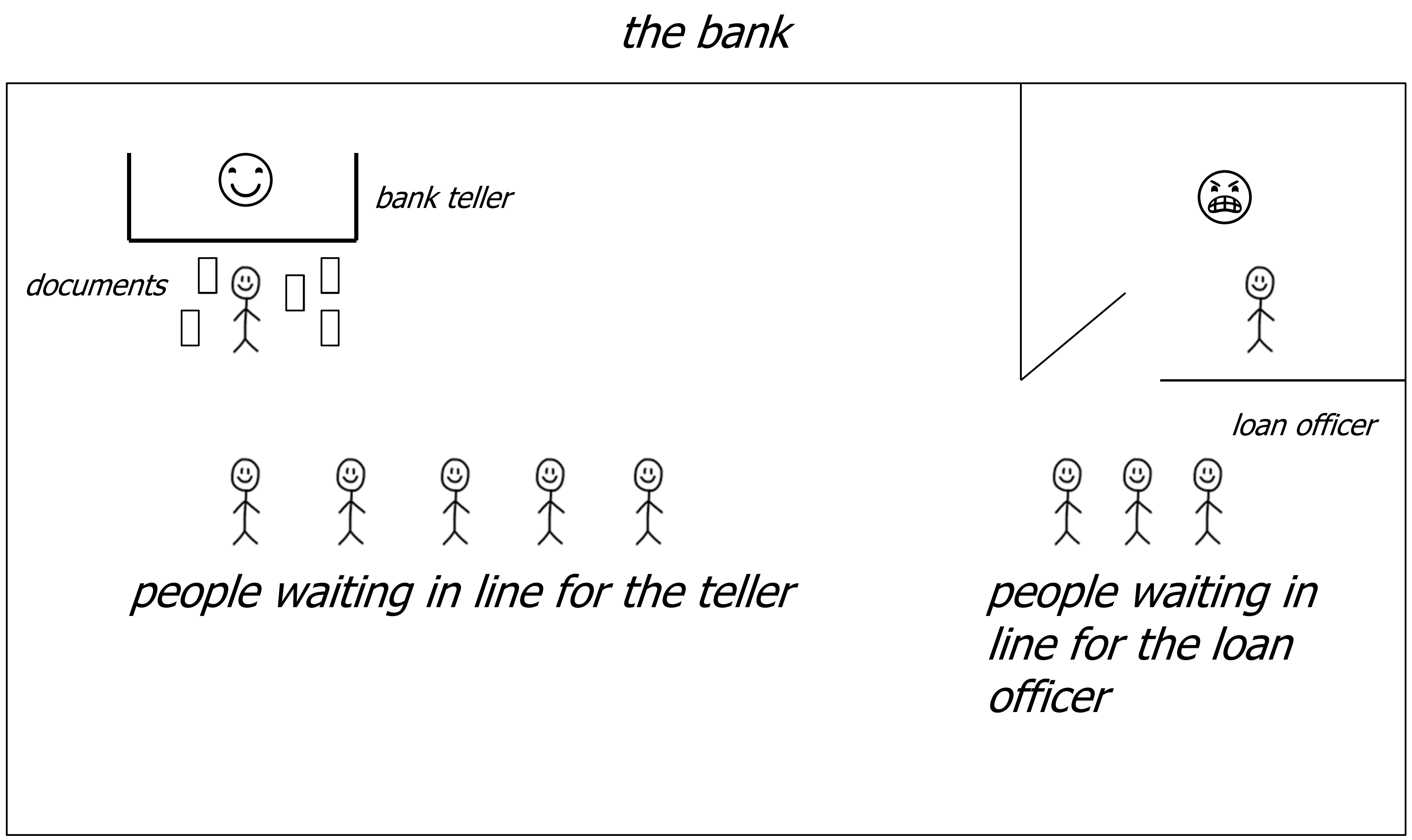
Process Scheduling

There is only one CPU, so who gets it?

- this is the question that process scheduling answers
- ideally, we want all processes to get the CPU periodically so that no processes *starve*
- and, we want the CPU to be busy as much of the time as possible

⇒ this sounds easy; it turns out that it's not so easy

Analogy: Bank Teller


the bank
bank teller
documents
loan officer
people waiting in line for the teller
people waiting in line for the loan officer

Supermarket Checkout Line

Which customer should go first?

Question *à la* iClicker

For my supermarket checkout line, I'm going to check out customers in the order in which they arrive in line.

With which statement do you most agree?

(a) this is the fairest way of getting everyone through the checkout line

(b) everyone will have to wait about the same period of time to get through the line

(c) people who have just a few items will not have to wait in line very long at all

(d) everyone will eventually get through the checkout line

get your iClickers out and fire them up!

Scheduling Queues

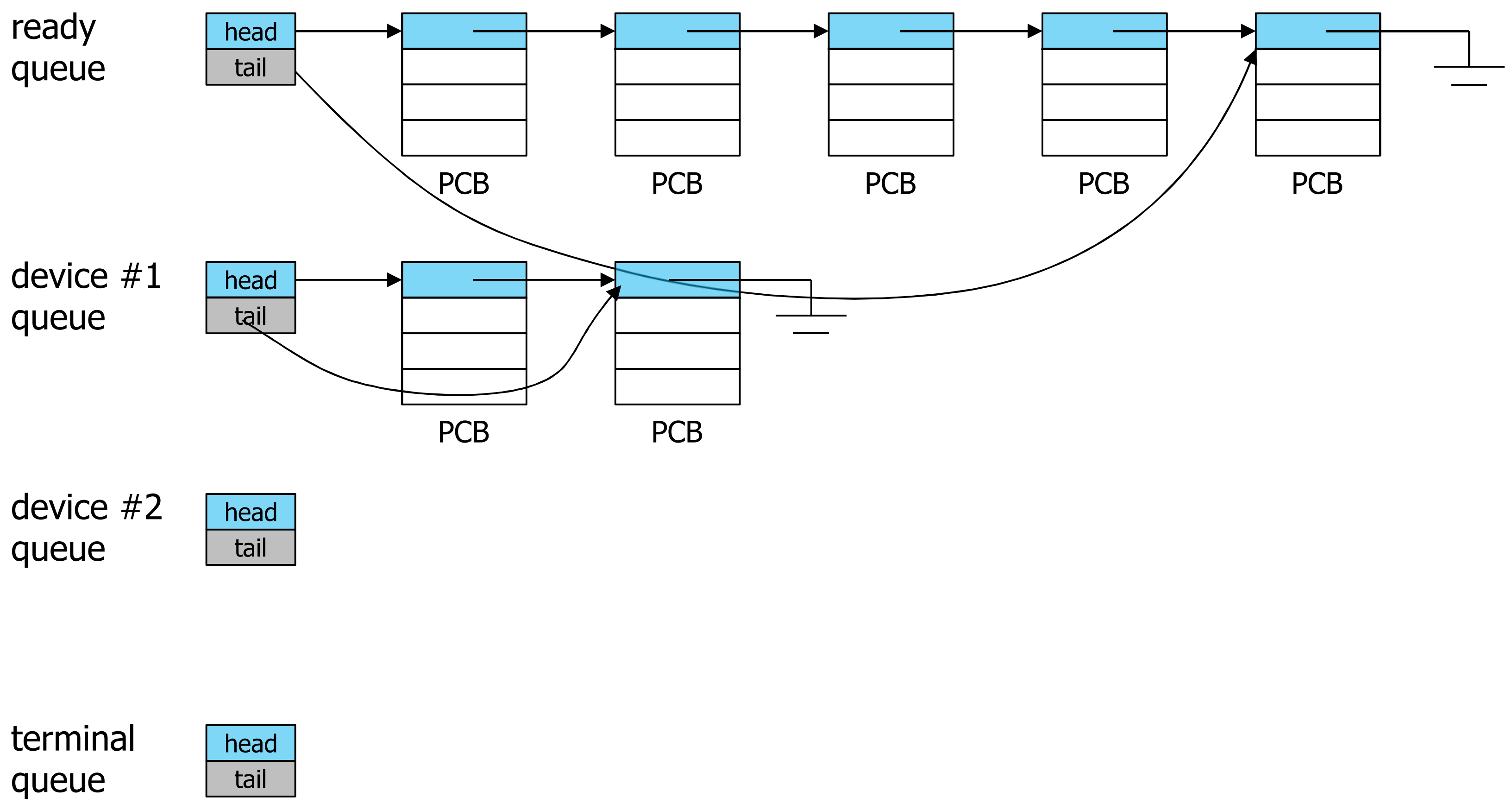
OS implements process scheduling using scheduling queues

- job queue, for new processes in the system
- ready queue, for processes that are ready to execute
- device queue, for processes that are waiting for particular device

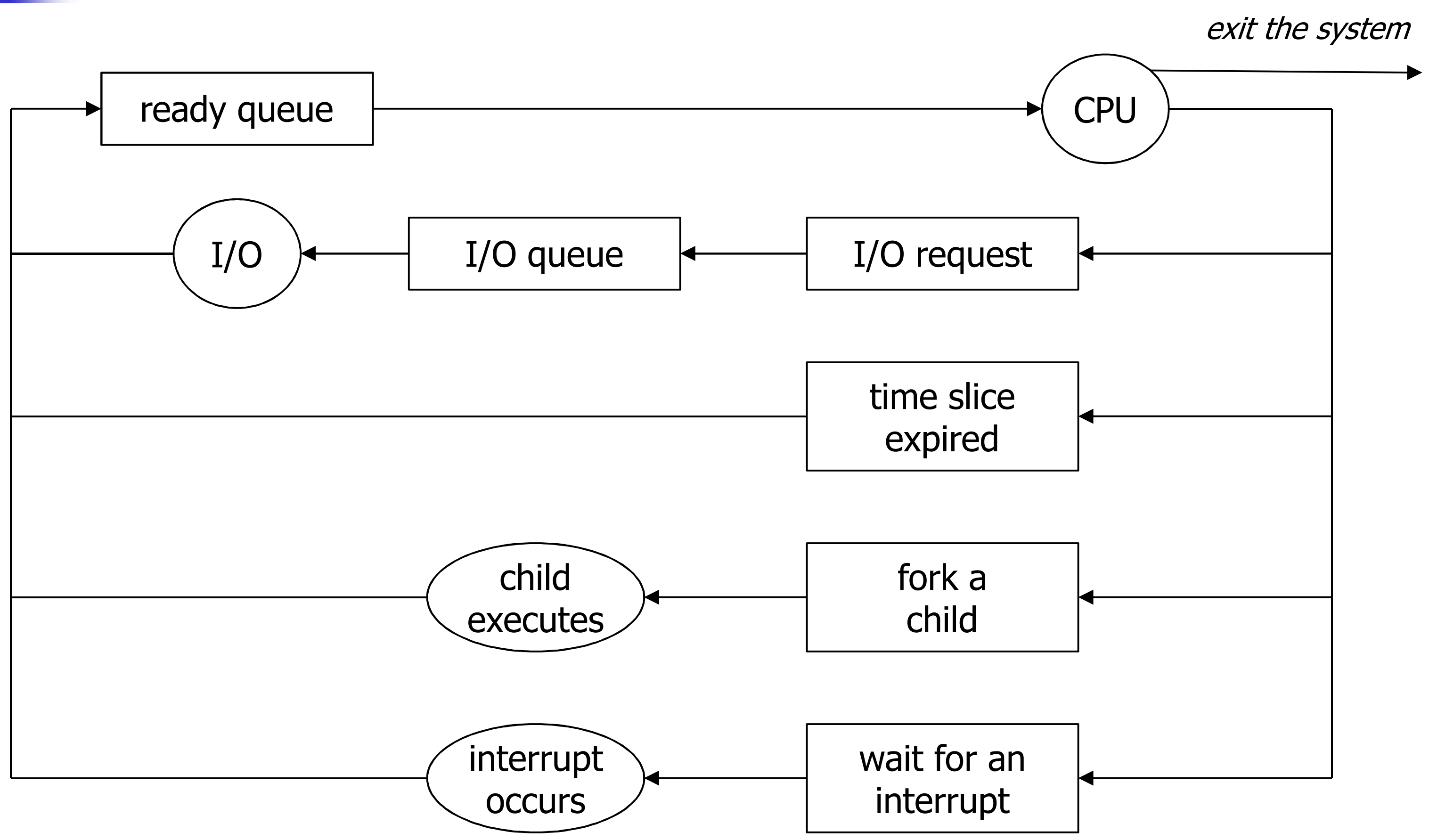
Remember that a queue is FIFO

- FIFO = first-in-first-out
- although there are variants such as priority queues

Scheduling Queues



Queueing Diagram



each box is a queue

Scheduling Queues

One way of managing system load:

- jobs that are submitted to the system go into an initial job pool
- the *long-term scheduler* periodically takes jobs from this job and puts them in the ready queue
- the *short-term scheduler* manages the dispatch to the CPU
- suppose the short-term scheduler makes a decision every 100 milliseconds
- how much processing can a job really accomplish in 100 milliseconds of CPU time?
- but, note that if it takes the OS 10 milliseconds to analyze the queues and make a decision about which process gets the CPU next, then this is a lot of "wasted" CPU overhead: 10 / (100 + 10) ≈ 9%

Alternative for Some Systems

The system might not have a long-term scheduler

- every job that is submitted goes into the ready queue
- what happens if too many jobs are submitted to the system?
- we count on self-regulation (just like in real life!)
- if there are too many jobs in the system, then the system will slow down, and some users will leave the system because it's too slow

Context Switch

Every time the OS needs to remove an active process from the CPU and dispatch a different process to the CPU, the OS performs a context switch

- save the state of the first process in its PCB
- load the state of the second process from its PCB
- this includes register contents, memory information, accounting information, etc.

This can be costly! A few milliseconds. (How long is a millisecond to us? How long is a millisecond to the CPU?)

Processes: a Few Basics

Each Linux process has a unique process ID (= pid)

When Linux boots up, it creates a process called “init”, with pid = 1

- there’s actually another important process in Linux called kthreadd that starts many kernel proceses

The init process is responsible for starting most of the other processes that run on the machine.

This leads to the concept of a process tree, with the init (pid=1) as the root.

ps -ef on a Linux system

```
UID        PID  PPID  C STIME TTY          TIME CMD
root         1     0  0 Jan02 ?        00:01:02 /sbin/init
root         2     0  0 Jan02 ?        00:00:00 [kthreadd]
root         3     2  0 Jan02 ?        00:00:00 [migration/0]
root         4     2  0 Jan02 ?        00:00:51 [ksoftirqd/0]
root         5     2  0 Jan02 ?        00:00:00 [stopper/0]
root         6     2  0 Jan02 ?        00:00:21 [watchdog/0]
root         7     2  0 Jan02 ?        01:36:54 [events/0]
root         8     2  0 Jan02 ?        00:00:00 [events/0]
root         9     2  0 Jan02 ?        00:00:00 [events_long/0]
root        10     2  0 Jan02 ?        00:00:00 [events_power_ef]
root        11     2  0 Jan02 ?        00:00:00 [cgroup]
root        12     2  0 Jan02 ?        00:00:03 [khelper]
root        13     2  0 Jan02 ?        00:00:00 [netns]
root        14     2  0 Jan02 ?        00:00:00 [async/mgr]
root        15     2  0 Jan02 ?        00:00:00 [pm]
root        16     2  0 Jan02 ?        00:00:59 [sync_supers]
root        17     2  0 Jan02 ?        00:00:02 [bdi-default]
root        18     2  0 Jan02 ?        00:00:00 [kintegrityd/0]
root        19     2  0 Jan02 ?        00:02:38 [kblockd/0]
root        20     2  0 Jan02 ?        00:00:00 [kacpid]
root        21     2  0 Jan02 ?        00:00:00 [kacpi_notify]
root        22     2  0 Jan02 ?        00:00:00 [kacpi_hotplug]
root        23     2  0 Jan02 ?        00:00:00 [ata_aux]
root        24     2  0 Jan02 ?        00:00:00 [ata_sff/0]
root        25     2  0 Jan02 ?        00:00:00 [ksuspend_usbd]
root        26     2  0 Jan02 ?        00:00:00 [khubd]
root        27     2  0 Jan02 ?        00:00:00 [kseriod]
root        28     2  0 Jan02 ?        00:00:00 [md/0]
root        29     2  0 Jan02 ?        00:00:00 [md_misc/0]
root        30     2  0 Jan02 ?        00:00:00 [linkwatch]
[+ about 50 process]
root      19954     1  0 Jun25 ?        00:00:02 /usr/sbin/sshd
root      24290     1  0 Jun12 ?        00:28:35 /var/cfengine/bin/cf-execd
root      24588     1  0 Jun12 ?        00:09:56 /var/cfengine/bin/cf-monitord
root      24702     1  0 Jun12 ?        00:44:32 /var/cfengine/bin/cf-serverd
root      28882 19954  0 09:00 ?        00:00:00 sshd: jhibbele [priv]
jhibbele 28893 28882  0 09:01 ?        00:00:00 sshd: jhibbele@pts/0
jhibbele 28894 28893  0 09:01 pts/0    00:00:00 -bash
jhibbele 31205 28894  0 09:13 pts/0    00:00:00 ps -ef
```

Process Creation

When a new child process starts, it needs certain resources

- CPU time
- memory
- access to files and I/O devices

A decision at the OS level is:

- how should the child be granted these resources?
- should the child (or children) have to share the resources that the parent process has?
- otherwise, child processes could end up hogging all of the system resources

Process Creation

Two specific questions:

(1) When a process creates a child process:

- should the parent continue to execute concurrently with its children?
- or should the parent wait for some or all of its children to terminate before it resumes execution?

(2) And also:

- should the child process be a duplicate of the parent, with the same program and data as the parent?
- or should the child process have a new program loaded into it?

Unix Process Creation

The system call `fork()` does the following:

- create a new process having a copy of the parent's address space
- start the child process
- return the pid of the new child

Now we have two copies of the same program running. Is this useful?

- Maybe.

Even more useful is to have one of the processes run a different program.

Unix man pages

```
$ man fork
FORK(2)                    Linux Programmer's Manual                    FORK(2)


NAME
       fork - create a child process

SYNOPSIS
       #include <unistd.h>

       pid_t fork(void);

DESCRIPTION
       fork()  creates  a new process by duplicating the calling process.  The
       new process, referred to as the child, is an  exact  duplicate  of  the
       calling  process,  referred  to as the parent, except for the following
       points:

       *  The child has its own unique process ID, and this PID does not match
          the ID of any existing process group (setpgid(2)).

       *  The  child's  parent  process ID is the same as the parent's process
          ID.

[etc.]
```

"man" stands for "manual"; try "apropos" also!

Unix exec()

`exec()` is the magic system call

`exec()` loads a new program into the current memory space of the process that calls it and starts this new program

This destroys the memory image of the process that calls the `exec()`

Unix Example: fork()

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

int main() {
  pid_t pid;
  int some_variable, some_other_variable;

  some_variable = 101;
  some_other_variable = 999;

  pid = fork();
  if (pid < 0) {
    fprintf(stderr, "fork failed\n");
    return(8);
  } else if (pid == 0) {
    printf("(C) I am the child\n");
    printf("(C) some_variable = %d, some_other_variable = %d\n",
           some_variable, some_other_variable);
    printf("(C) sleep(4)\n");
    sleep(4);
    execlp("/bin/ls", "ls", NULL);
  } else {
    wait(NULL);
    printf("(P) I am the parent -- child is complete\n");
    printf("(P) some_variable = %d, some_other_variable = %d\n",
           some_variable, some_other_variable);
  }
  return(0);
}
```

```
$ gcc fork-example.c
$ a.out
(C) I am the child
(C) some_variable = 101, some_other_variable = 999
(C) sleep(4)
abort.c  a.out  fork.c  list.c  sigsegv.c
(P) I am the parent -- child is complete
(P) some_variable = 101, some_other_variable = 999
```

MS Windows

The system call CreateProcess() wants (among other things) the name of a specific program to load and start.

WaitForSingleObject() waits for a single child process to terminate.

Process Termination

When does a process terminate?

- it calls `exit()` to tell the OS that it’s done
- the child attempts to do something bad (improper)
- the child exceeds the usage of some of the resources that were allocated to it
- some other process kills the child (usually, only an “ancestor” process can kill a child)
- the parent process of this process is exiting, and the OS does not allow a child to continue if its parent terminates

This last one leads to an interesting situation in Unix.

Unix exit() and wait()

In Unix, an exiting process passes an integer return code to the OS.

For example, `exit(4)`

The OS—or the parent process—can then take action based on the value of that return code.

The parent process can examine the child's return code using `wait()`

Unix exit() and wait()

Here is how the parent checks the return code of the child:

```
pid_t pid;
int status;

pid = wait(&status);
if (status != 0) {
  // non-zero value means the child returned a non-zero
  // value: parent can take action
}
```

What happens if the child exits but the parent isn't waiting for it? What happens to the return-code information?

Zombie Process

Zombie Process

In Unix, a process that ends becomes a zombie.

But only until the parent `waits` for the process, which should be a brief time.

If the parent process terminates before the child process ends. then the child becomes a (zombie) child of the init process (pid == 1, the parent of all processes in the system).

And the init process periodically does a `wait()` itself.

Interprocess Communication

Some processes can do their work independently of other processes—they don’t need to communicate with any other processes (except the OS).

Some processes require cooperation with other processes in the system.

Cooperating Processes

Processes might cooperate for various reasons:

- information sharing: two (or more) processes want to access a shared resource (such as a file) – why do we need cooperation?
- computation speedup: distribute the work among several processes and let them work on the task in parallel – what must be true about the system in order for this to provide a speed-up?
- modularity: break a complex system into pieces, and let different processes work on different tasks
- convenience: allow a user to work on many tasks at the same time

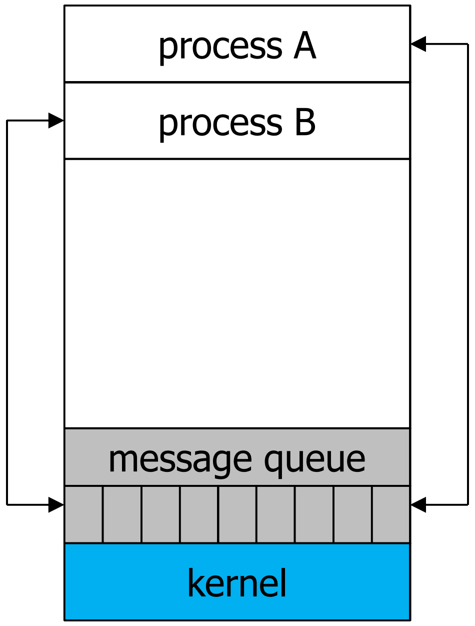
Good example is a browser: essentially one process for each tab (why is this a good idea?)

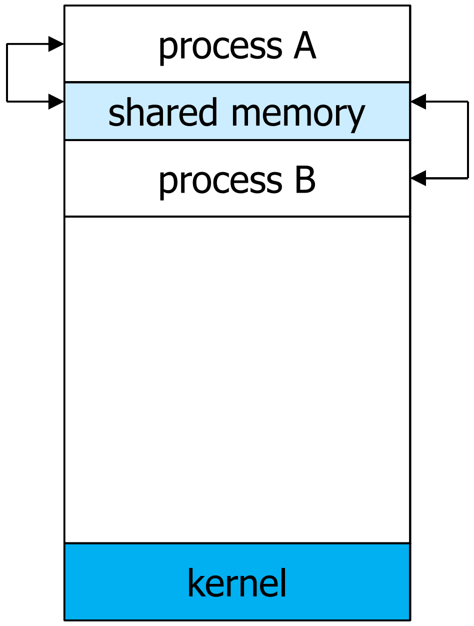
Interprocess Communication (IPC)

There are two basic frameworks for IPC

- message passing
- shared memory

both schemes require synchronization!





Message Passing

Simplest requirement is to have two operations

```
send(message)
receive(message)
```

But, the interesting thing is in the details:

- send, but send to whom?
- receive from whom?
- how to send to more than one process?
- how much can I send (does the system have a limit)?
- how big is a message? Fixed size? Variable?

Message Passing

Here are general characteristics of message passing

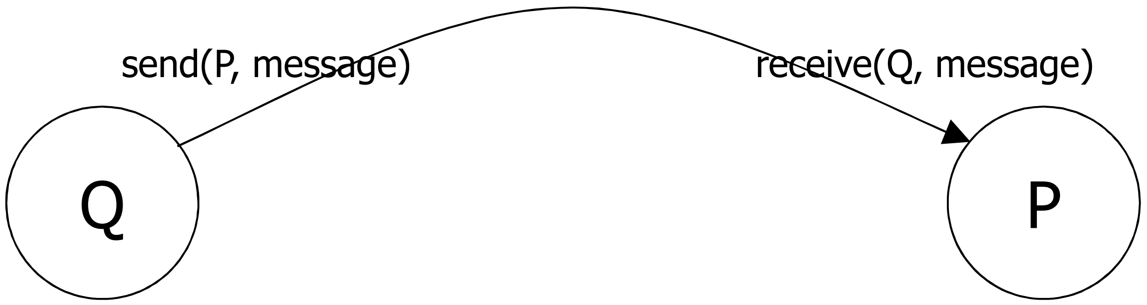
- direct vs. indirect communication
- synchronous vs. asynchronous communication
- automatic vs. explicit buffering

Symmetric Message Passing

Each (sender, recipient) names the other in the message-passing call

send(P, message): send message to process P

receive(Q, message): receive message from process Q



Symmetric Message Passing

Each (sender, recipient) names the other in the message-passing call

send(P, message): send message to process P

receive(Q, message): receive message from process Q

Requirements:

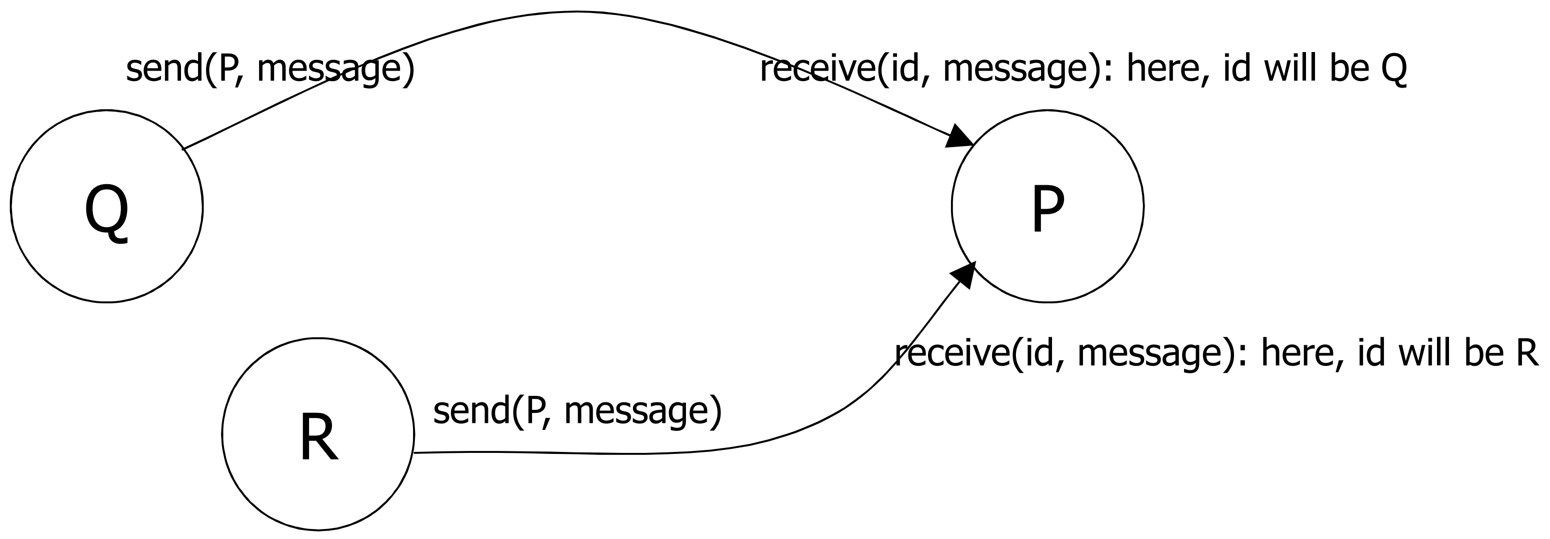
- link must exist between P and Q
- the link is associated ONLY with P and Q
- P and Q have only one link associated with them

Asymmetric Message Passing

Only the sender names the recipient

send(P, message): send message to process P

receive(id, message): receive a message; id is set the name of the sender

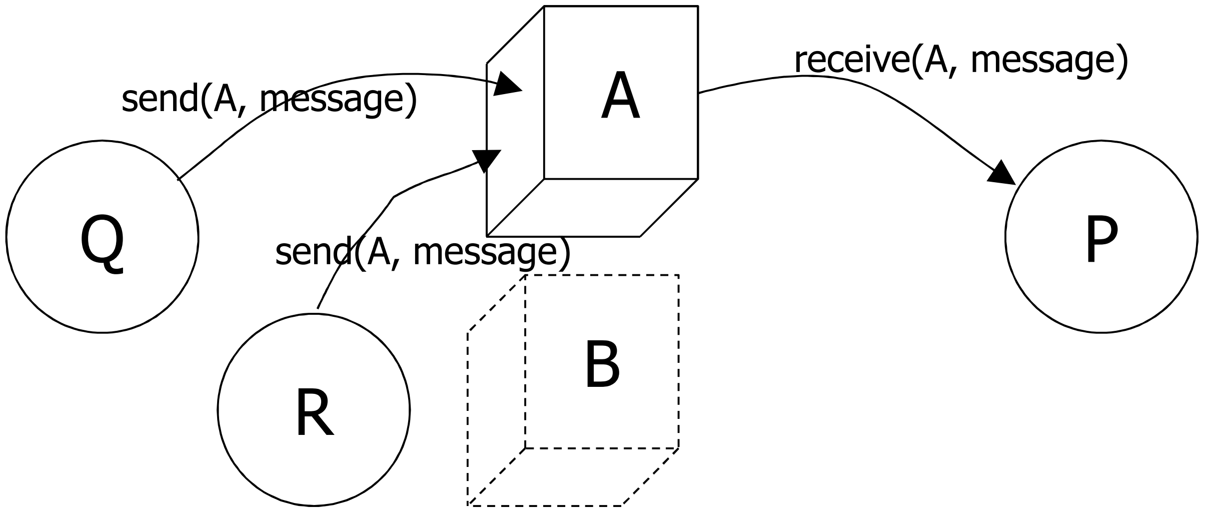

send(P, message)
receive(id, message): here, id will be Q
Q
P
R
receive(id, message): here, id will be R
send(P, message)

Indirect Communication

Use a “mailbox”, to which two or more processes have access:

send(A, message): send message to mailbox A

receive(A, message): receive a message from mailbox A



Indirect Communication

Properties of this kind of communication

- both processes need a link (access to the mailbox)
- a link can be associated with more than just two processes
- more than one link (i.e., mailbox) can exist between two process

So, we’d say this is more general.

Synchronization

How do sender and recipient coordinate their actions?

- blocking send: sending process is blocked (i.e., it waits) until the recipient (or the mailbox) receives the message
- nonblocking send: sending process sends the message and continues its execution
- blocking receive: recipient blocks until a message is available
- nonblocking receive: recipient retrieves either a valid message or null

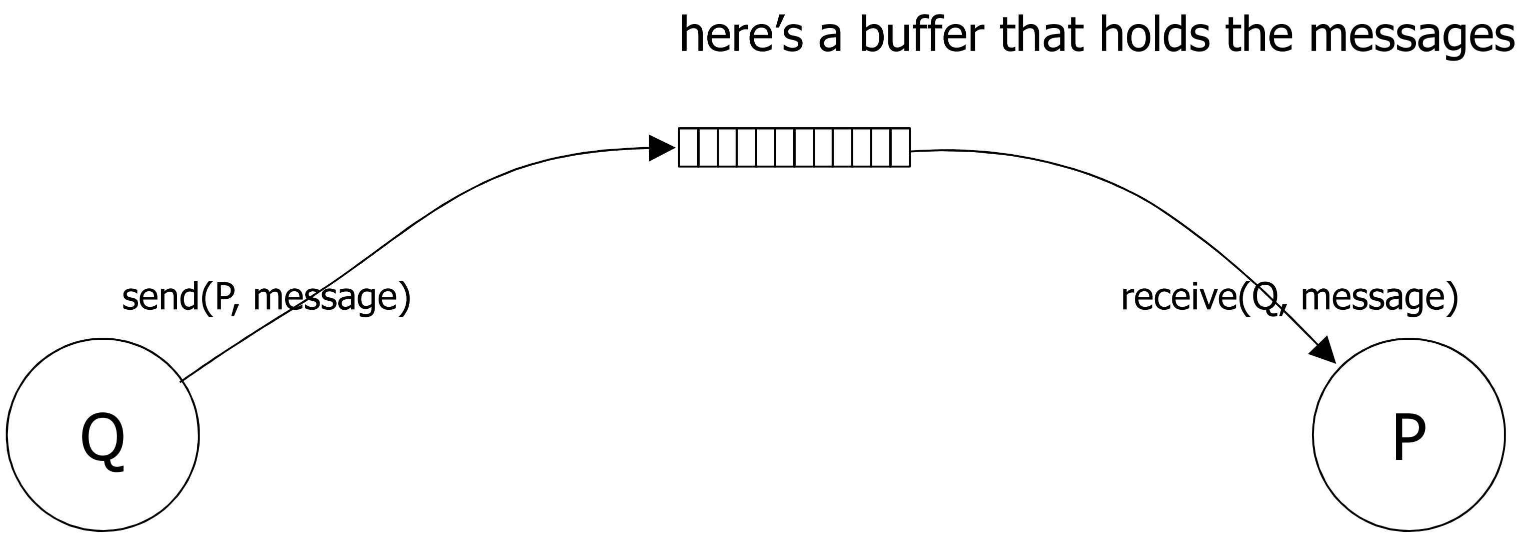
Synchronization

Blocking send/receive is an easier paradigm for the application developer, but it can be inefficient.

Nonblocking send/receive raises another issue: buffering.

Buffering

Message queue



If we have a buffer to hold messages, what questions does this raise?

Buffering

In a message-passing system, there is a queue somewhere.

This queue can be implemented in different ways

- queue has effective length of zero: implications?
- queue has bounded capacity; what happens if the queue is full?
- queue has unbounded capacity: implications?

Buffering

In a message-passing system, there is a queue somewhere.

This queue can be implemented in different ways

- queue has effective length of zero, which means that sender must block until recipient receives the message
- queue has bounded capacity; if the queue is full then sender must block
- queue has unbounded capacity: sender never needs to block

Shared Memory for a Buffer

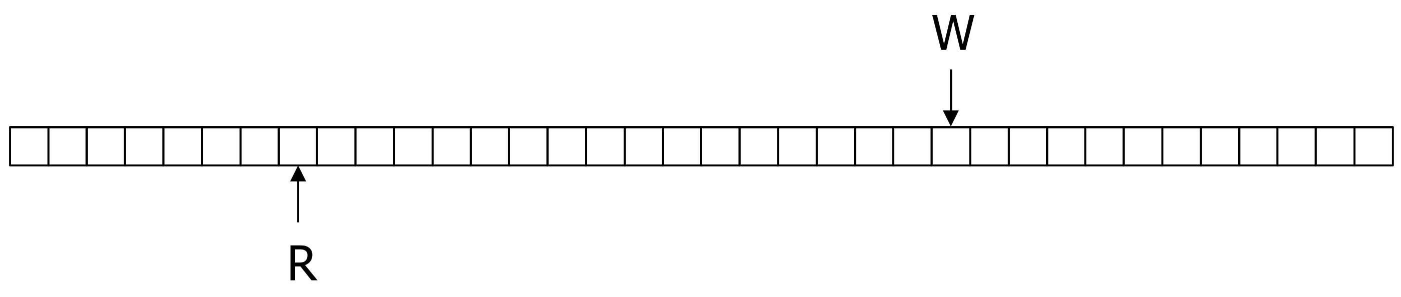
Conceptually, very simple

- OS makes a block of memory available that > 1 process can access
- the memory might live in the address space of the process that creates it, but the actual implementation isn't particularly important
- the important thing is that more than one process can access the memory

Shared Memory: Producer/Consumer

Here's the block of memory, with two pointers (or indices)

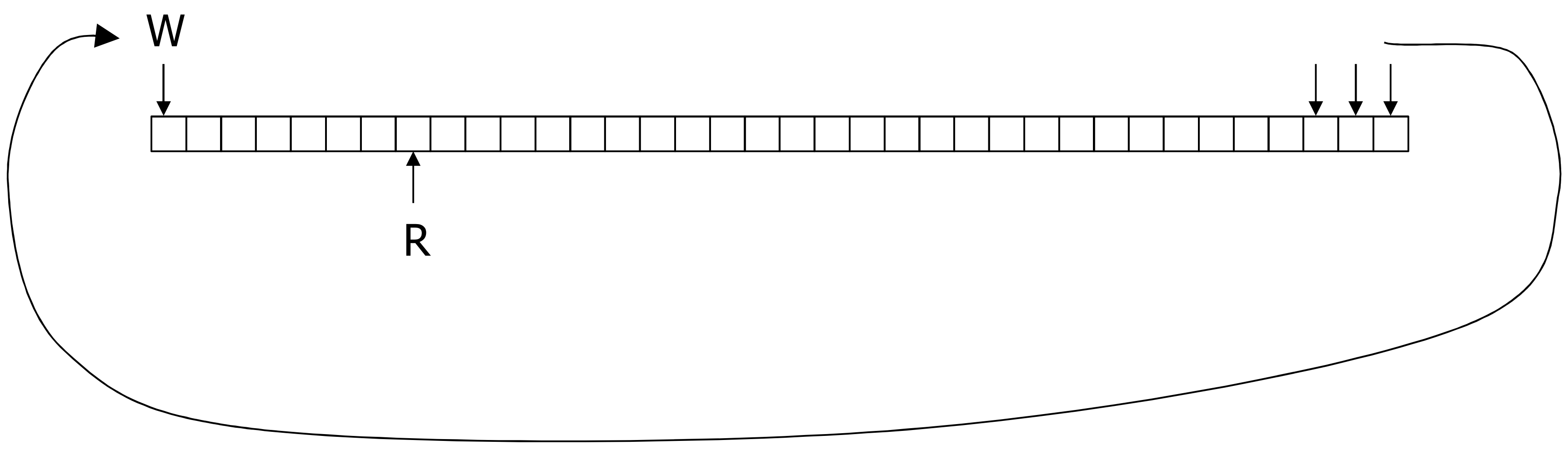
- W is where producer writes; R is where consumer reads



We can implement this with a *circular buffer*.

Circular Buffer

Here's a circular buffer

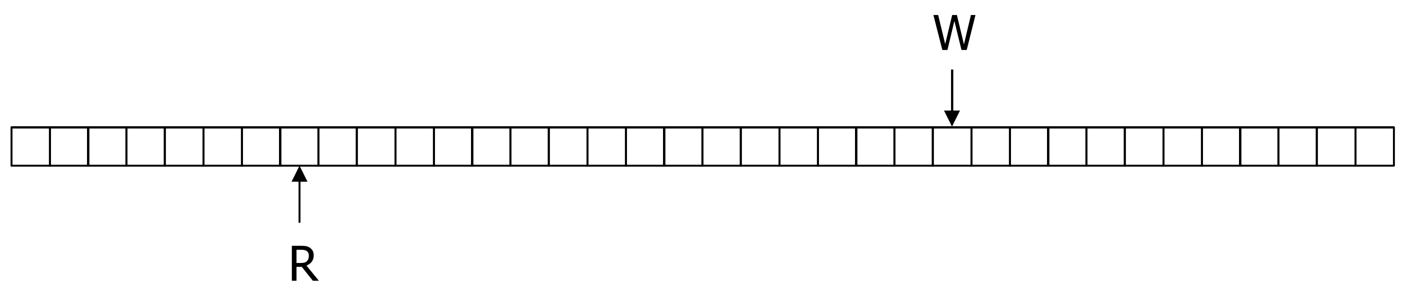


when the writer gets to the end, it wraps around

Shared Memory: Producer/Consumer

Circular buffer

- W is where producer writes; R is where consumer reads



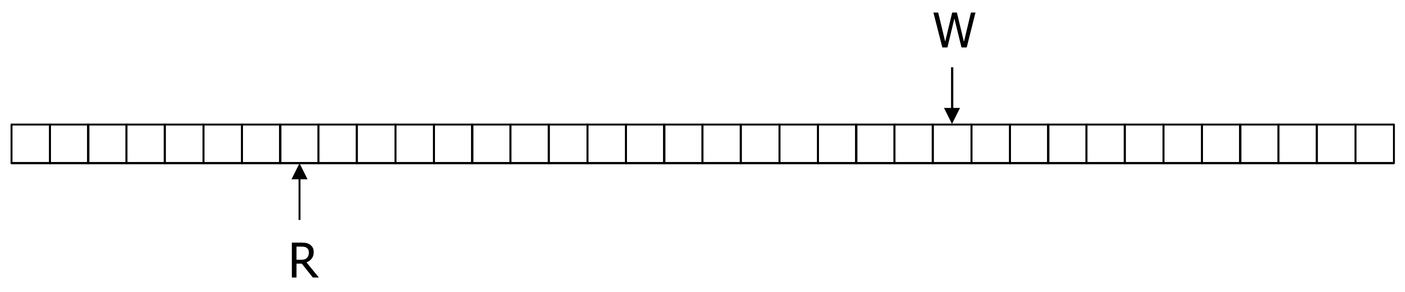
Then, there are some things we must think about

- what are the considerations?

Shared Memory

Considerations:

- If the buffer is bounded in size, then what happens when the queue is full?
- How do we prevent producer and consumer from accessing the same data simultaneously?



Another Communication Mechanism

Socket: abstraction of a network facility enabling communication between different processes

- possibly even on different machines

Easiest to see an example from Java

Java Example

```
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) {
    String hostname = “kaladin.cems.uvm.edu";
    //String hostname = “127.0.0.1”;
    int portnum = 6013;

    try {
      Socket sock = new Socket(hostname, portnum);
      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);
    }
  }
}
```

- Webservers listen on port 80
- This program (the server) is listening on port 6013
- IP address 127.0.0.1 is called “the loopback” and means the local machine
- my example is running (and listening) on kaladin

Remote Procedure Calls (RPC)

Formal protocol for allowing a client to execute a function on a server

- using structured messages
- and a fixed and published description of how messages correspond to server functions
- so for example we could implement this using sockets
- the server would listen for messages and parse them
- and then perform actions based on the contents of the message

Pipes

An abstraction of an interprocess communication facility to a file

- so that a process can write to the “file” and a different process can read from the “file”

An example of this in UNIX is the “|” (pipe) operator:

```
$ ls | less
$ ps -ef | grep jhibbele
```