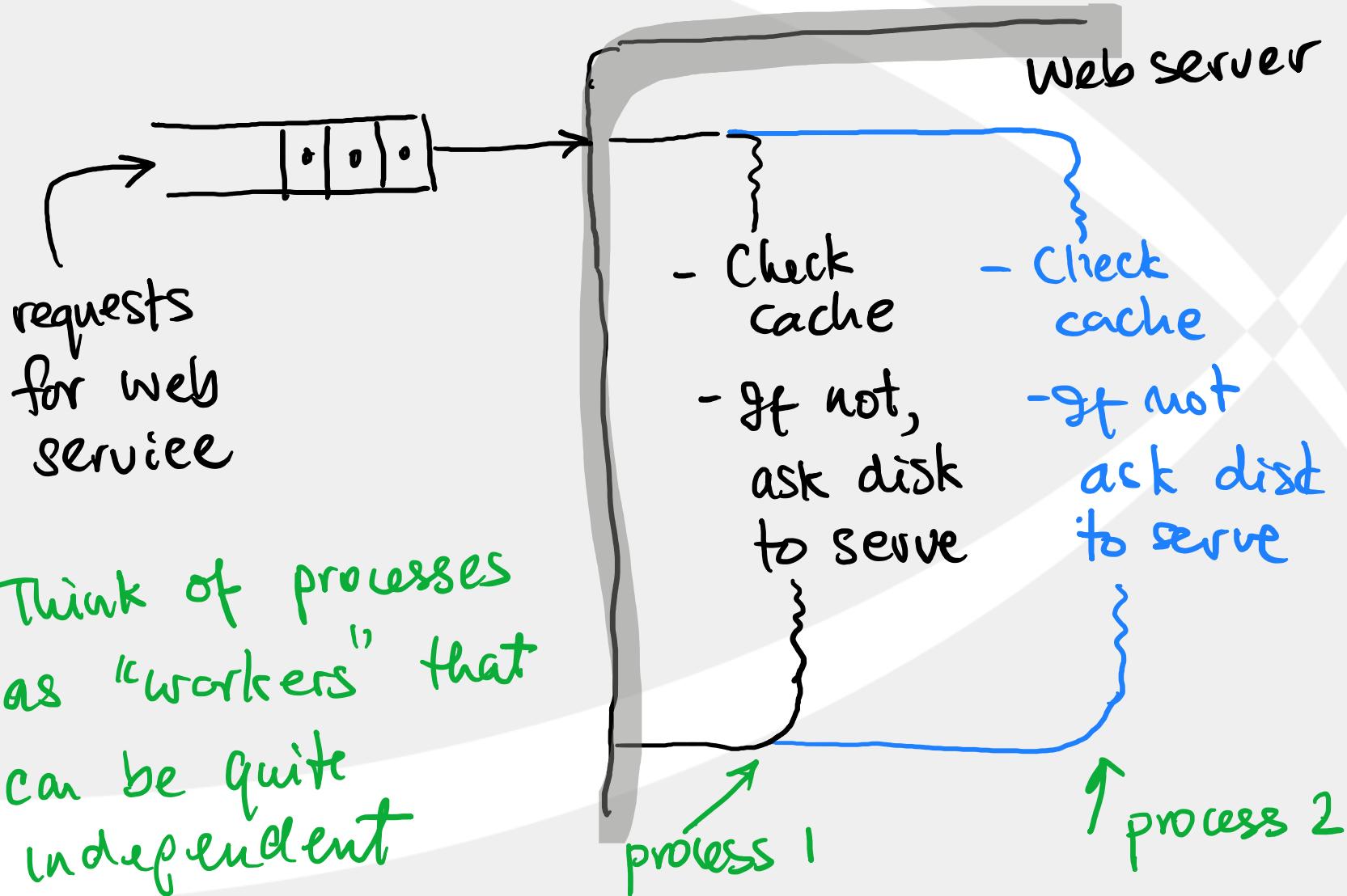


# Processes and Threads

# What is a *process*?

- Process
  - An abstraction of a running program
  - Modern OS hinges on this concept
  - Supports the ability to have (pseudo) concurrent operation
- To illustrate, lets consider a web server

# Scenario: Web Server



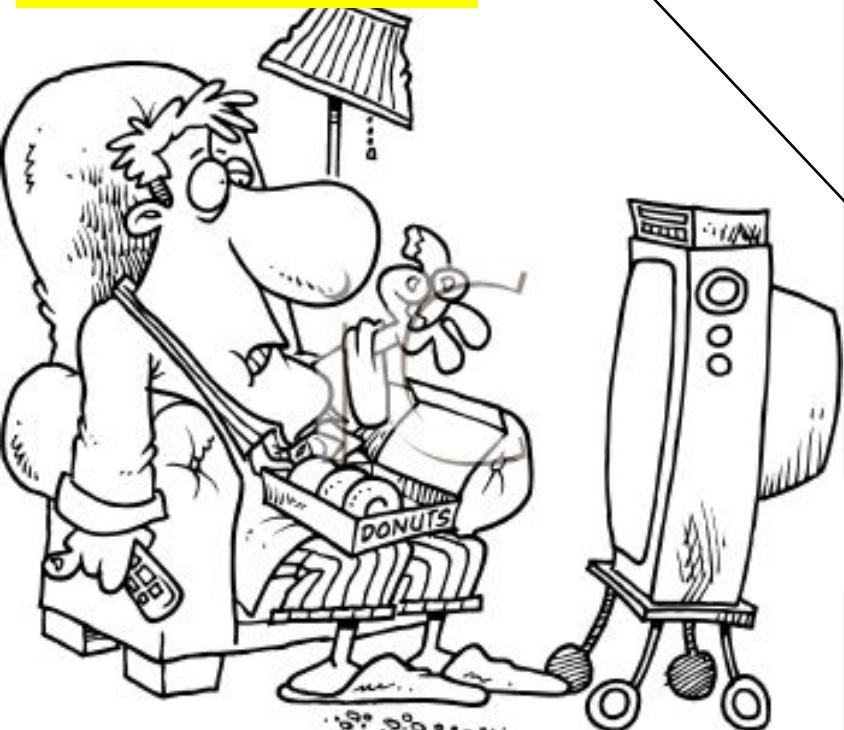
# Analogy from the Book



"Process" is the activity of baking the cookies

# Analogy from the Book...

“Process” is the activity of watching football



“Process” is the activity of baking the cookies

Single person needs to switch between the two different processes..

# Key Elements of a Process

points to the currently executing statement of the program

program counter

registers

input data

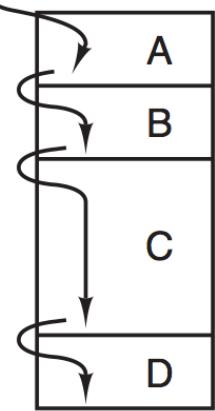
program

some key elements missing... what are they?

# Multiprogramming

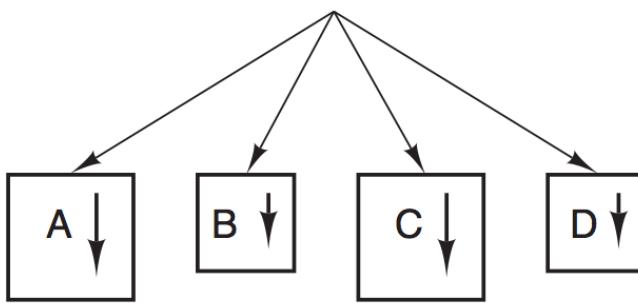
One program counter

Process switch

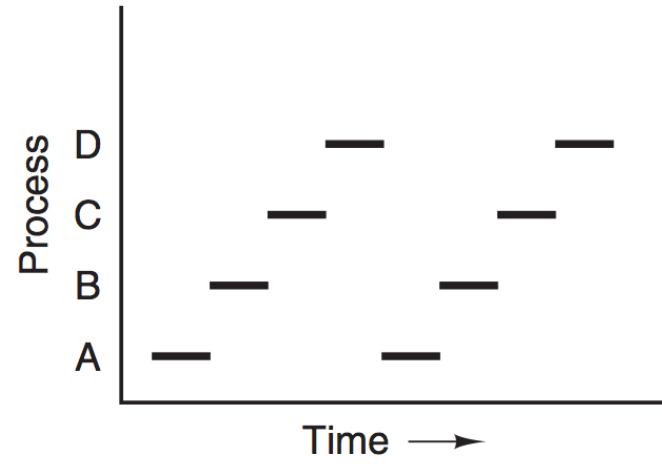


(a)

Four program counters



(b)



(c)

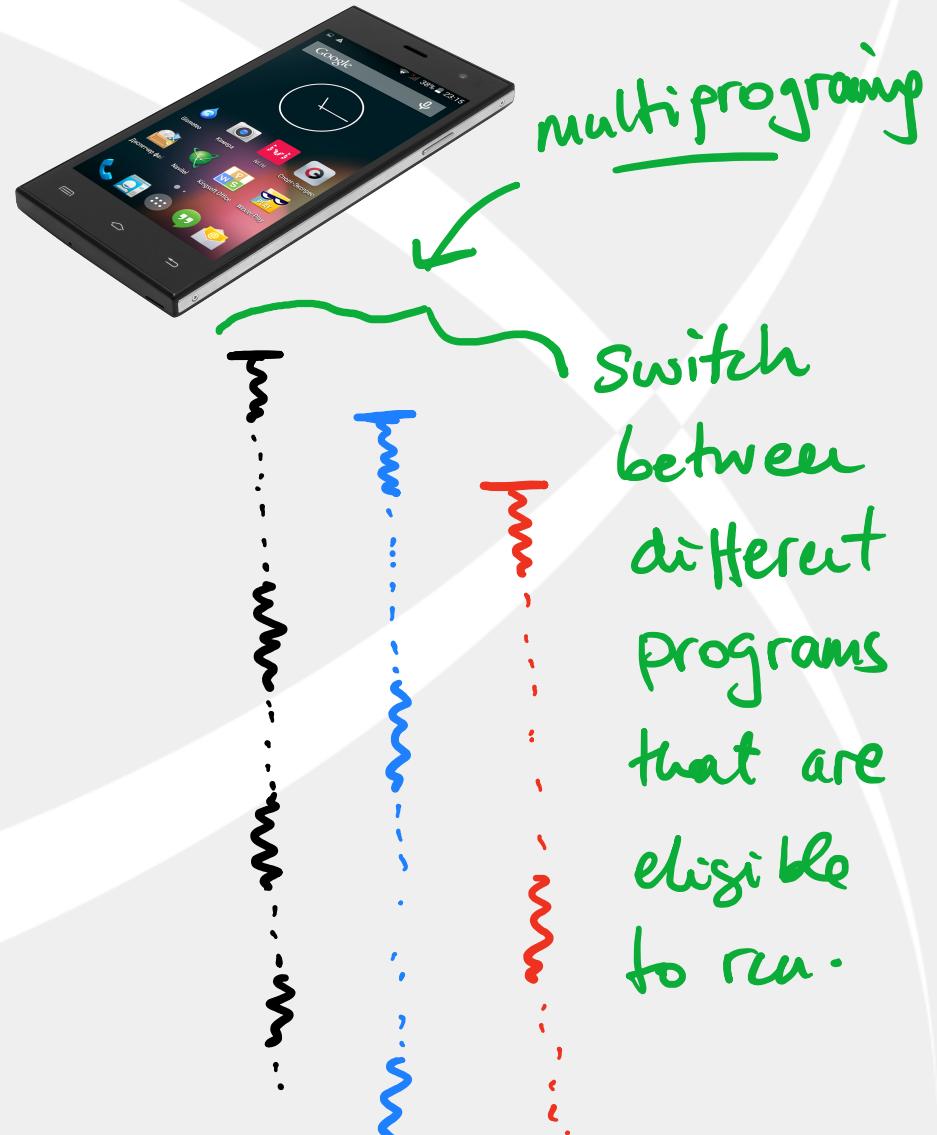
**Figure 2-1.** (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

Write the pseudo code for the “program” switching; what do we need to save and restore?

# Uniprogramming vs. Multiprogramming



Runs a single program at a time

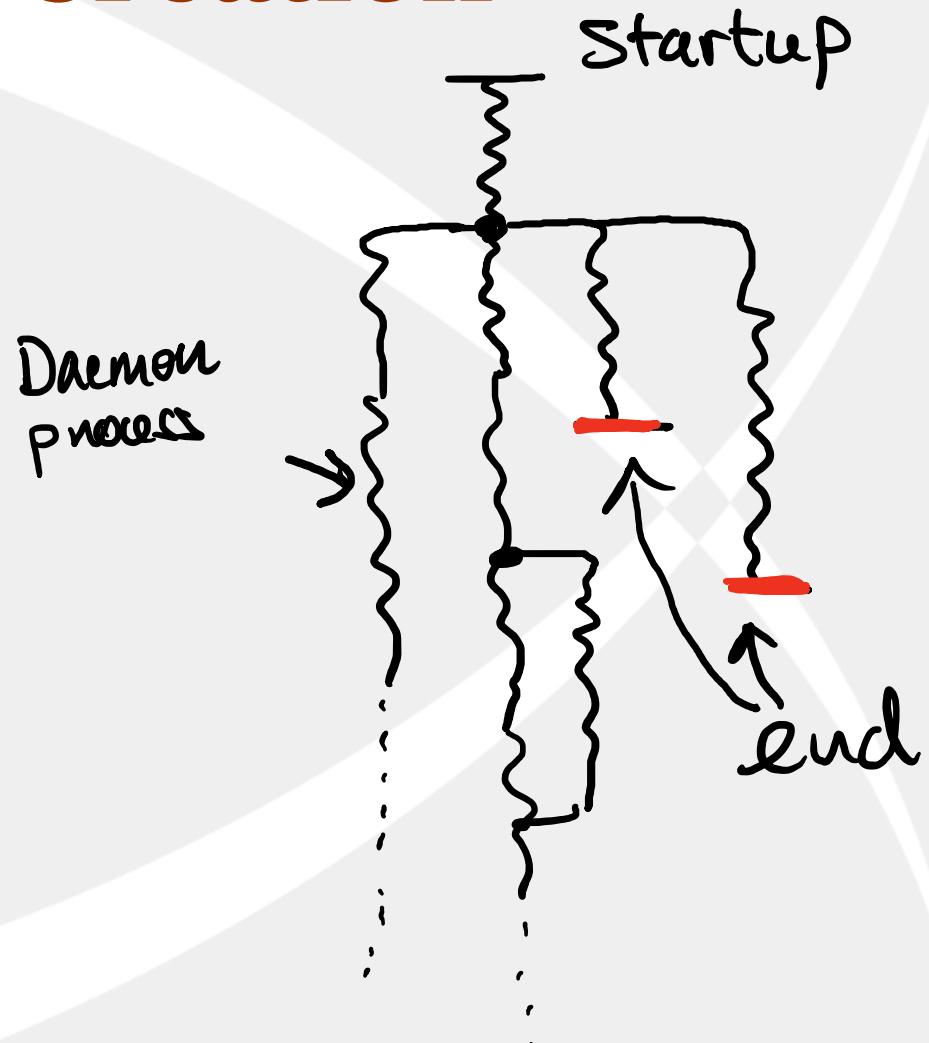


# Process management issues?

- Process management issues:
  - ◆ Lifecycle management
  - ◆ Precedence management (flow management)
- Lifecycle management:
  - ◆ Process creation
  - ◆ Process state changes, reasons (what happens in the middle!)
  - ◆ Process termination

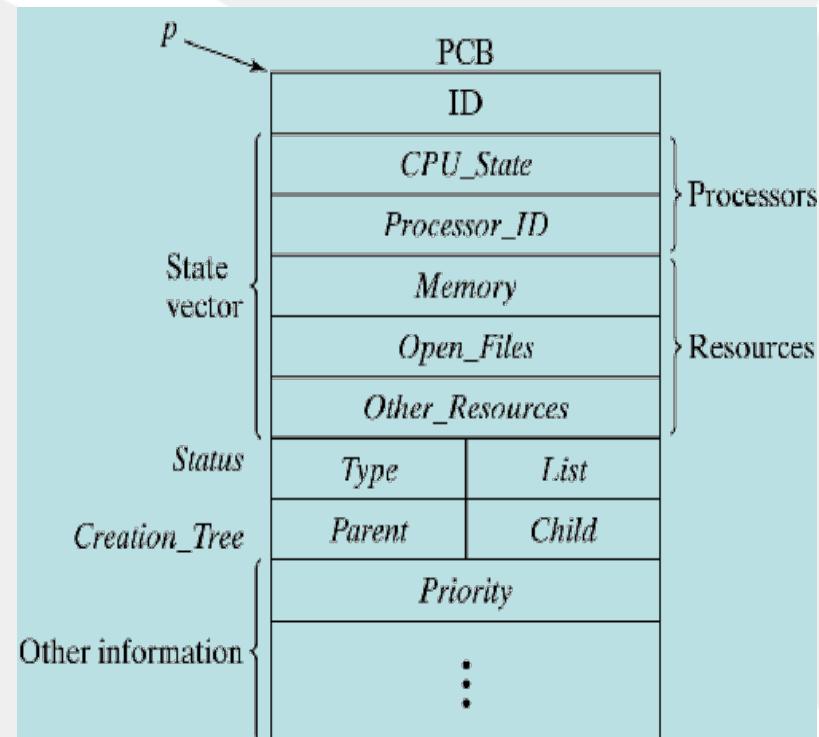
# Process Creation

- Four principle events creating processes
  - System initialization
  - Execution of a process creation system-call
  - User request to create a new process
  - Initiation of a batch job



# How is a process represented?

- Information: state & control
- Process Control Block (PCB)
  - ◆ Identifier
  - ◆ State Vector = Information necessary to run process p
  - ◆ Status
  - ◆ Creation tree
  - ◆ Priority
  - ◆ Other information



# Lifecycle: Create process

## ■ Two ways of creating a new process:

- ❖ *Build one from scratch:*

- load *code* and *data* into memory
- create (empty) a *dynamic memory workspace (heap)*
- create and initialize the *process control block*
- make process known to process scheduler (dispatcher)

Windows  
approach

- ❖ *Clone an existing one:*

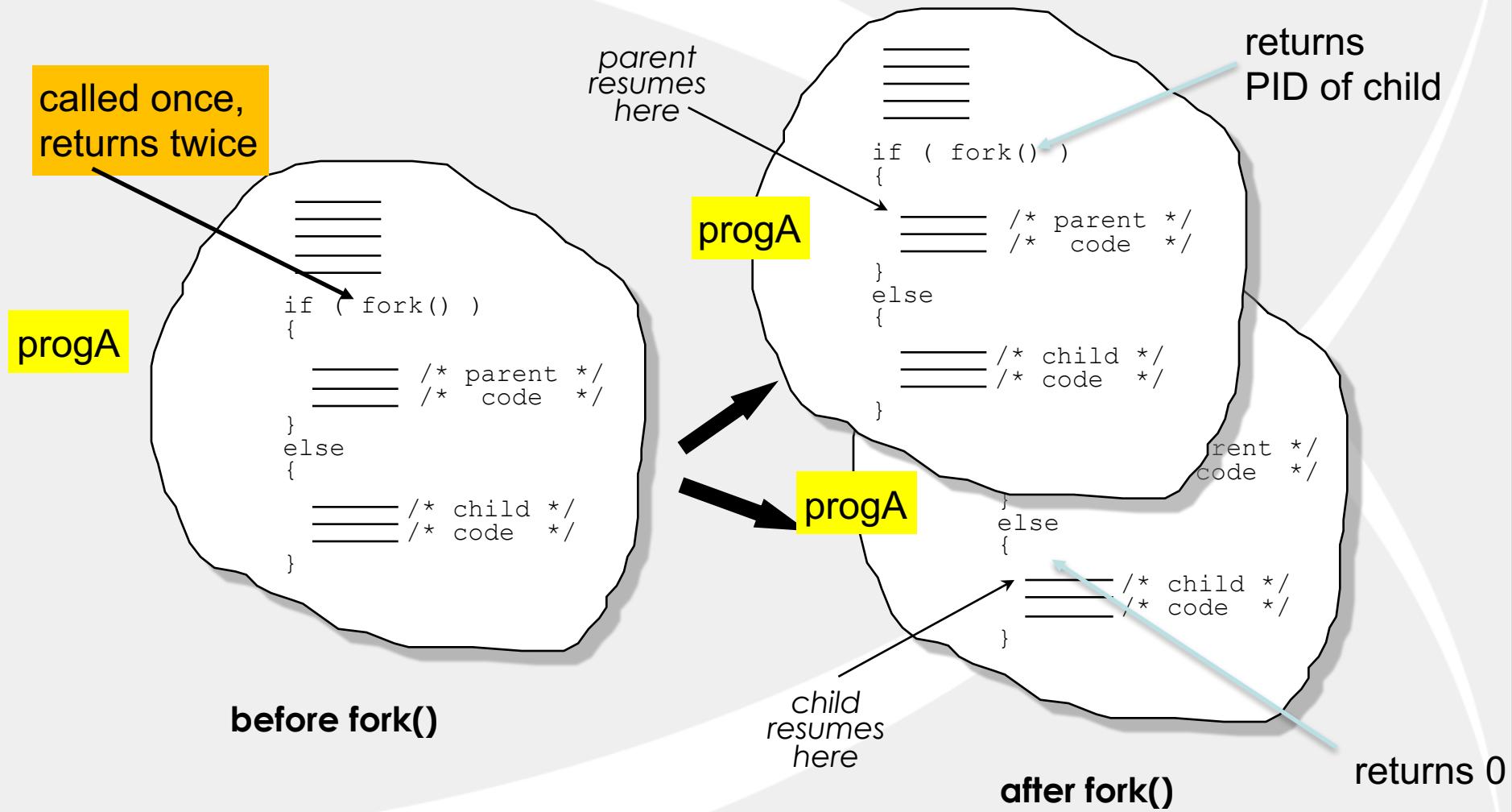
- stop current process and save its state
- make a copy of *code*, *data*, *heap* and *process control block*
- make process known to process scheduler (dispatcher)

UNIX  
approach

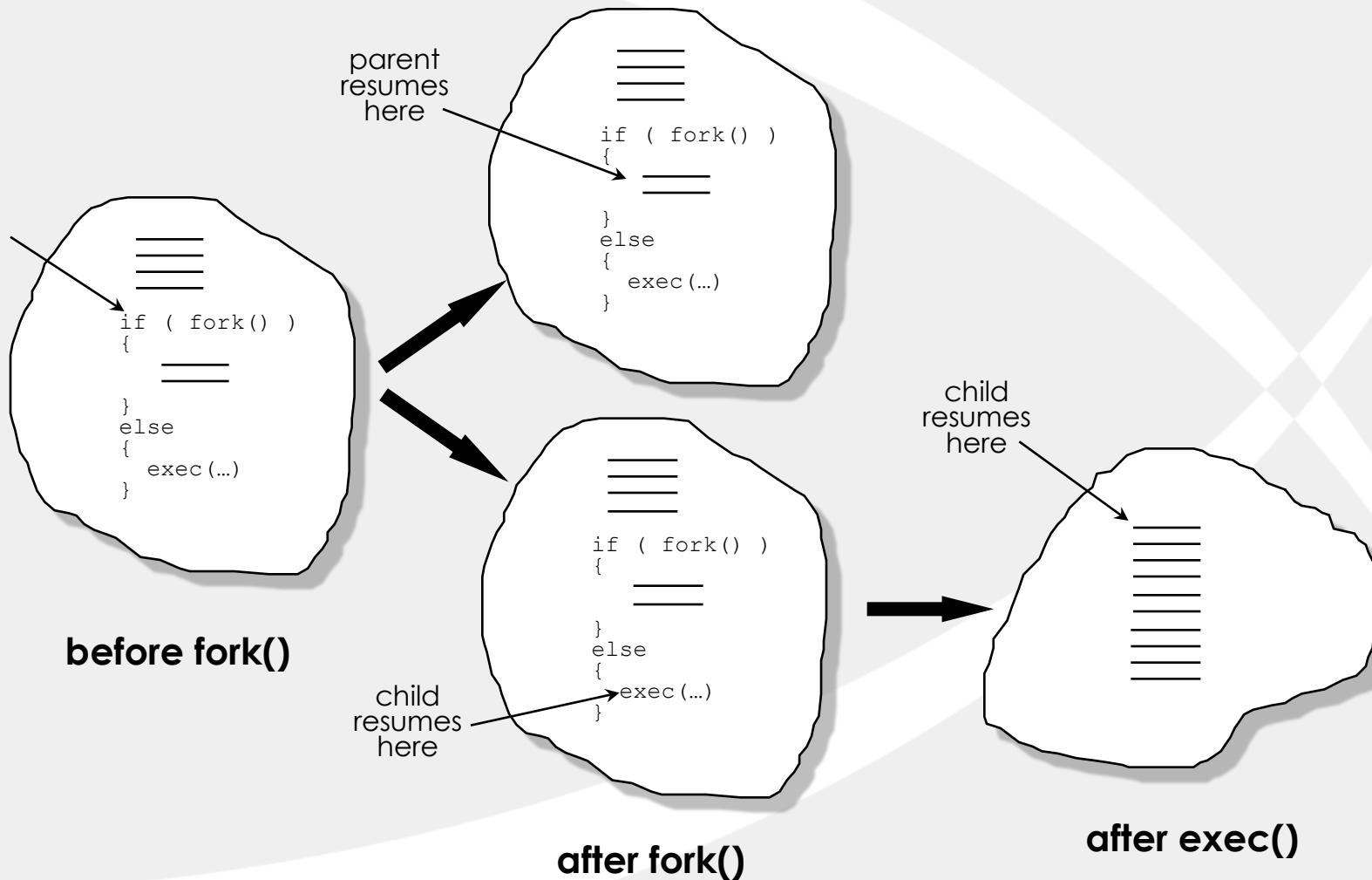
# UNIX process creation

- In UNIX, the **fork()** system call is used to create processes
  - ◆ **fork()** creates an identical copy of the calling process
  - ◆ after the **fork()** , the *parent* continues running concurrently with its *child* competing equally for the CPU

# UNIX process creation...



# A typical use of fork()



# Example

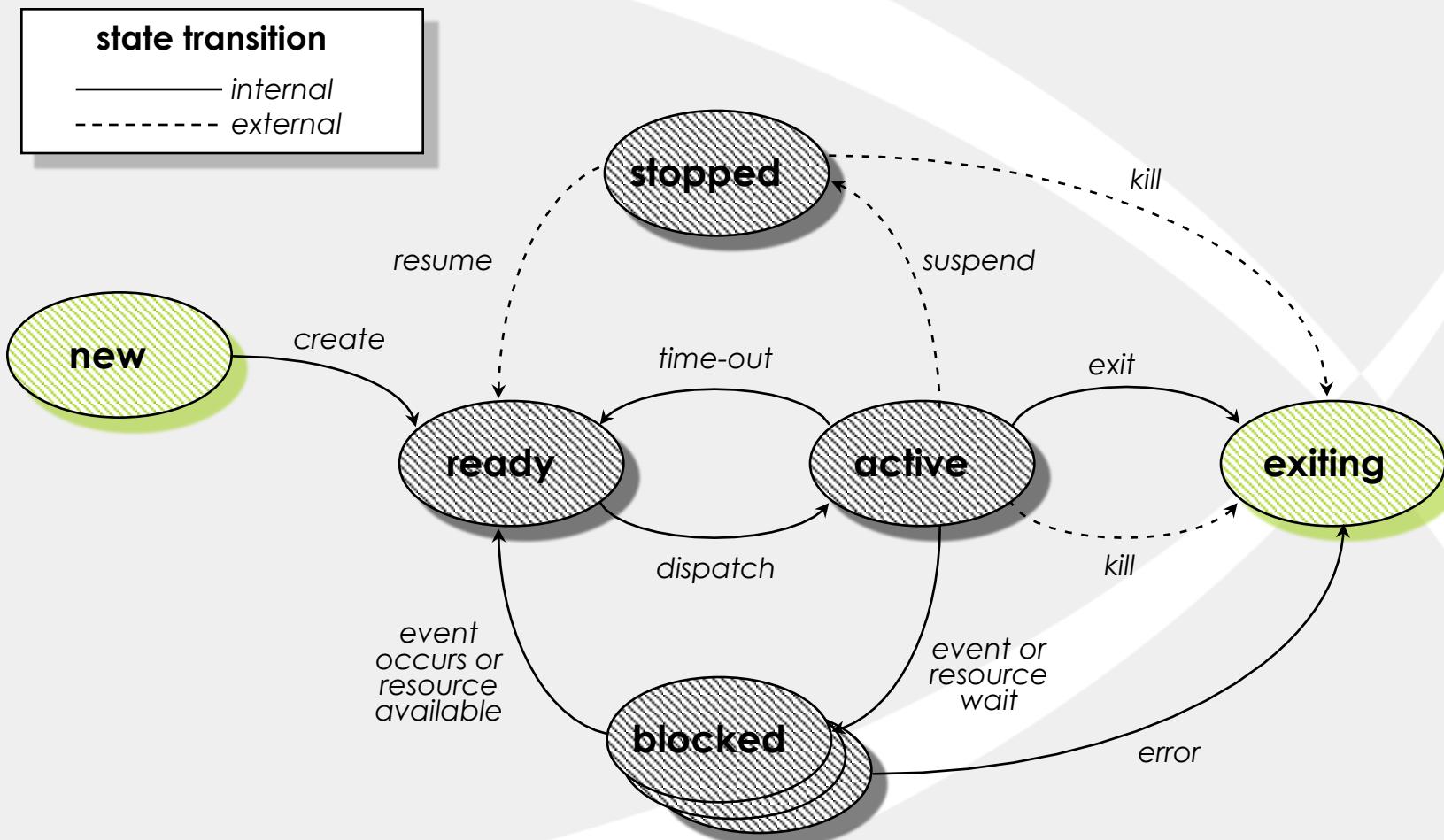
- What is the output of the following simple C program?

```
main() {  
    int i;  
    i = 10;  
    if (fork() == 0) i += 20;  
    printf(" %d ", i);  
}
```

# Lifecycle: After creation

- After creation, process can experience various conditions:
  - ◆ No resources to run (e.g., no processor, memory)
  - ◆ Waiting for a resource or event
  - ◆ Completed the task and exit
  - ◆ Temporarily suspend waiting for a condition
- → Process should be in different states

# Process state diagram



# Process states

- A process can be in many different states:
  - ❖ **New**—a process being created but not yet included in the pool of executable processes (*resource acquisition*)
  - ❖ **Ready**—processes are prepared to execute when given the opportunity
  - ❖ **Active**—the process that is currently being executed by the CPU
  - ❖ **Blocked**—a process that cannot execute until some event occurs
  - ❖ **Stopped**—a special case of **blocked** where the process is suspended by the operator or the user
  - ❖ **Exiting**—a process that is about to be removed from the pool of executable processes (*resource release*)

# Example

- Following are code segments from a process that is already created and eligible to run or running

```
....  
(a)  i = i + j * 10;  
     a[i] = b[j] * c[i];  
(b)  read(scale);      // reading standard  
     input  
           for a variable  
....  
(c)  wait (mutex);    // waiting on a mutual  
     exclusion variable
```

- What are the possible process states at (a), (b), and (c)?

# Lifecycle: Process termination

- A process enters the *exiting* state for one of the following reasons
  - ◆ normal completion: A process executes a system call for termination (e.g., in UNIX `exit()` is called).
  - ◆ abnormal termination:
    - programming errors
      - *run time*
      - I/O
    - user intervention

# Tiny Shell

- Repeat

- ◆ Show a prompt
  - ◆ Read a line of input
  - ◆ Run the given command in a child process

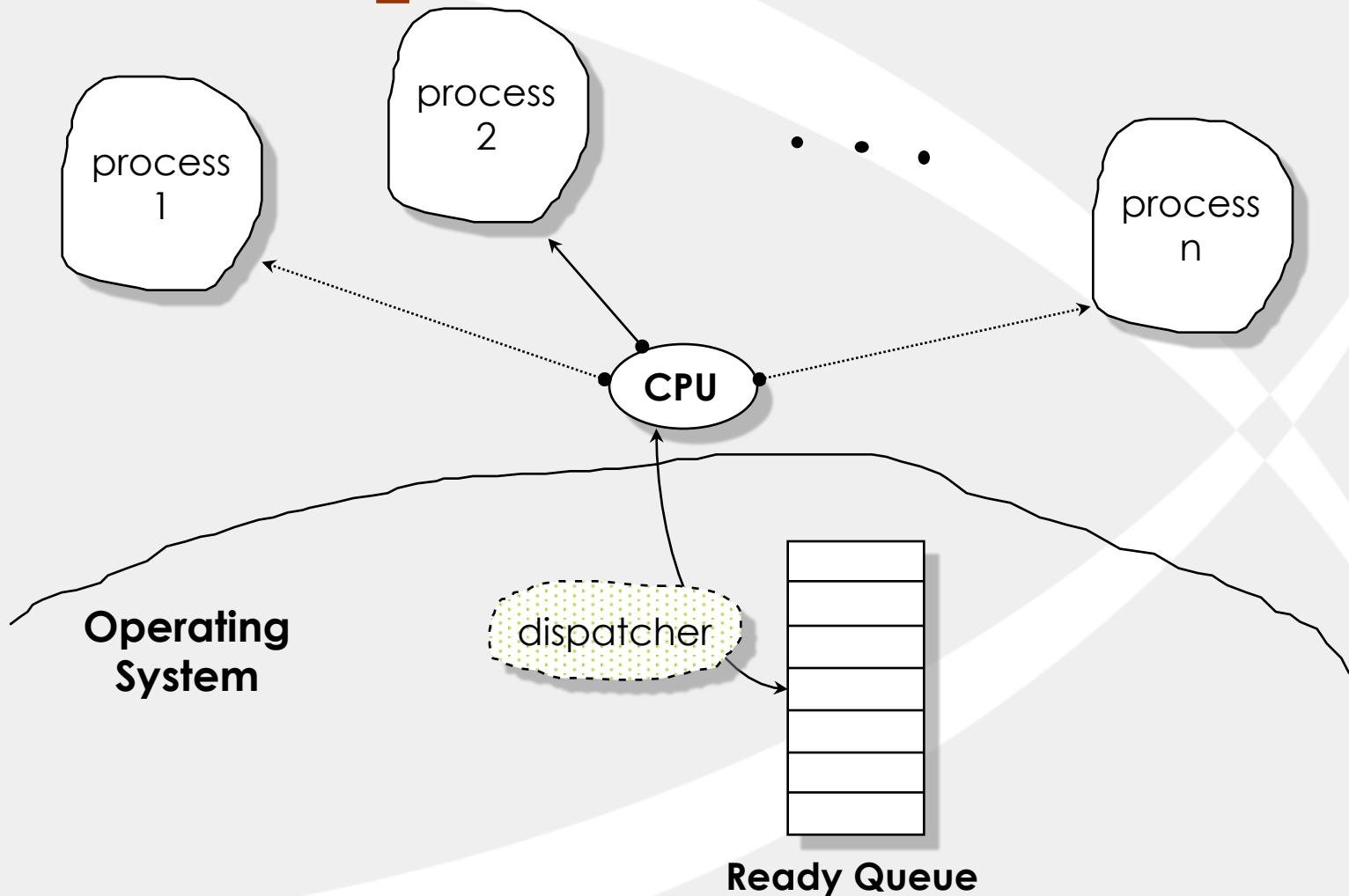
```
while (1) {  
    printf("Shell > ")  
    getline(line)  
    if (strlen(line) > 1) {  
        if (fork() == 0) {  
            exec(line)  
        }  
        wait(child)  
    }  
}
```

# Implementing processes

- With multi-programming, we have several processes concurrently executing
- OS is responsible:
  - Dynamically selecting the next process to run
  - Rescheduling performed by the dispatcher
- Dispatcher given by:

```
loop forever {
    run the process for a while.
    stop process and save its state.
    load state of another process.
}
```

# Dispatcher at work



# Dispatcher: Controlling the CPU?

- CPU can only do one thing at a time
- While user process running, dispatcher (OS) is NOT running
- How does the dispatcher regain control?
  - ◆ Trust the process to wake up the dispatcher when done (*sleeping beauty approach*).
  - ◆ Provide a mechanism to wake up the dispatcher (*alarm clock*).
- Obviously, the *alarm clock* approach is better. Why?

# How is an alarm event handled?

- Context switch happens:
  - ❖ OS saves the state of the *active* process and restores the state of the *interrupt service routine*
  - ❖ Simultaneously, CPU switches to *supervisory mode*
- What must get saved? *Everything that the next process could or will damage.* For example:
  - *Program counter (PC)*
  - *Program status word (PSW)*
  - *CPU registers (general purpose, floating-point)*
  - *File access pointer(s)*
  - *Memory (perhaps?)*
- While saving the state, the operating system should mask (disable) *all* interrupts.

# Memory: *to save or NOT to save*

- Here are the possibilities:

- ◆ Save *all* memory onto disk.

Could be *very* time-consuming. E.g., assume data transfers to disk at 1MB/sec. How long does saving a 4MB process take?

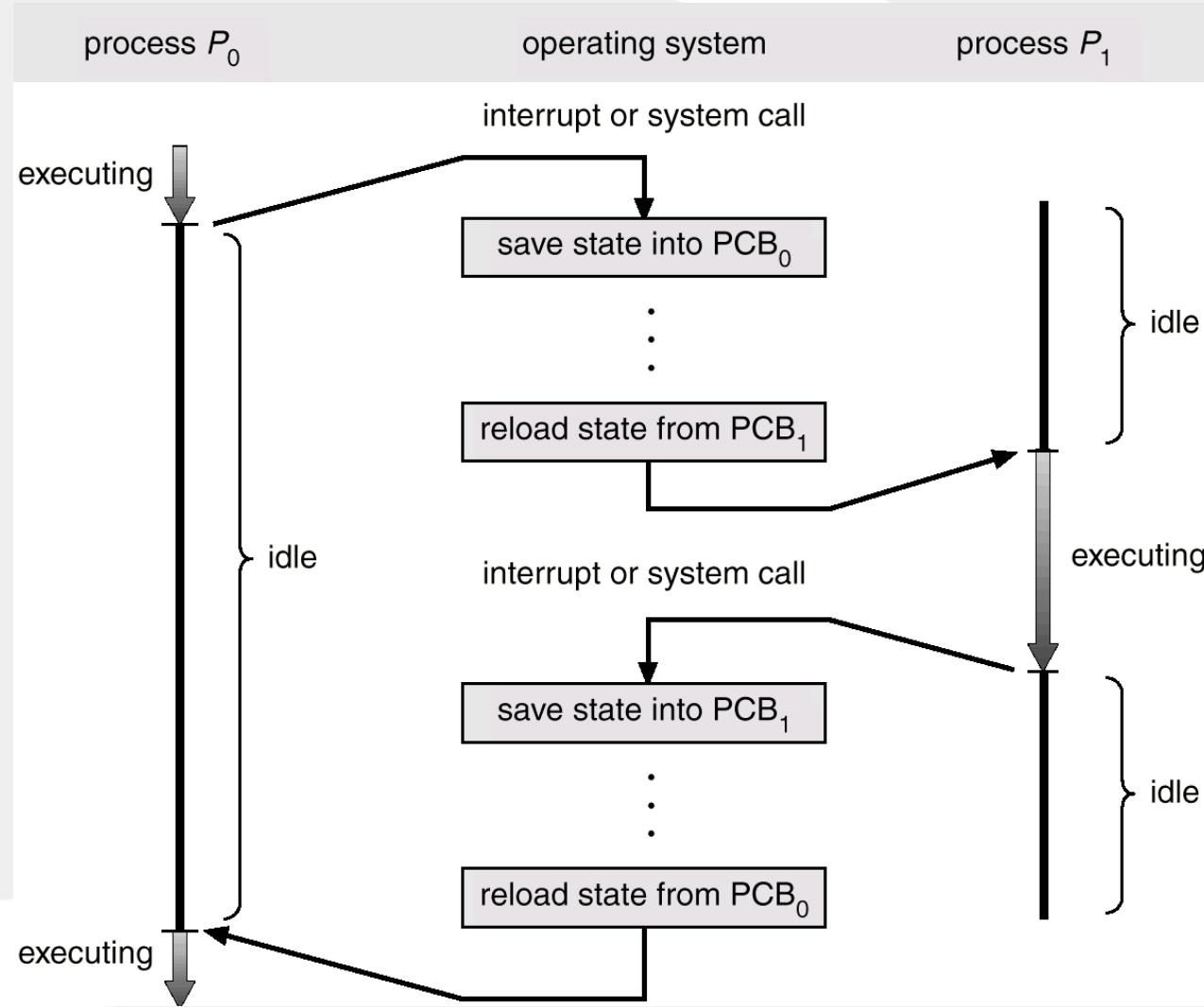
- ◆ Don't save memory; trust next process.

This is the approach taken by (older) PC and Mac OSes.

- ◆ Isolate (protect) memory from next process.

This is *memory management*, to be covered later

# CPU switching among processes



# Synchronization

# Concurrent Processes

- Concurrent processes can:
  - ◆ Compete for shared resources
  - ◆ Cooperate with each other in sharing the global resources
- OS deals with competing processes
  - ◆ carefully allocating resources
  - ◆ properly isolating processes from each other
- OS deals with cooperating processes
  - ◆ providing mechanisms to share resources

# Processes: *Competing*

- Processes that do not work together cannot affect the execution of each other, but they can compete for devices and other resources
- **Example:** Independent processes running on a computer
- **Properties:** Deterministic; reproducible
  - ❖ Can stop and restart without side effects
  - ❖ Can proceed at arbitrary rate

# Processes: *Cooperating*

- Processes that are aware of each other, and directly (by exchanging messages) or indirectly (by sharing a common object) work together, may affect the execution of each other
- **Example:** Transaction processes in an airline reservation system
- **Properties:**
  - ◆ Share a common object or exchange messages
  - ◆ Non-deterministic; May be irreproducible
  - ◆ Subject to *race conditions* – coming up!

# Why Cooperation?

- Cooperation clearly presents challenges – why do we want it?
- We may want to share resources:
  - ❖ Sharing a global task queue
- We may want to do things faster:
  - ❖ Read next block while processing current one; divide a job into pieces and execute concurrently
- We may want to solve problems modularly

UNIX example:

```
cat infile | tr ' ' '\012' | tr '[A-Z]'  
'[a-z]' | sort | uniq -c
```

See next slide for an alternate explanation

## Problem with

### Original application..

- Hard to understand or executing programs
- Instructions of the program arbitrarily
  - ❖ For cooperating processes, the order of (some) instructions are irrelevant
  - ❖ However, certain instruction combinations must be avoided, for example:

```
A = 1  
B = 2  
A = B + 1  
B = B * 2
```

<u>Process A</u>	<u>Process B</u>	<u>concurrent access</u>
$A = 1;$	$B = 2;$	<i>does not matter</i>
$A = B + 1;$	$B = B * 2;$	<i>important!</i>

Consider a sequential  
program fragment

$A = 1$

$B = 2$

$A = B + 1$

$B = B * 2$

Suppose  
we are  
interested  
in concurrently  
executing  
this program  
fragment.

Using concurrent  
execution we  
could exploit  
hardware  
parallelism to  
run faster where  
appropriate

→ For correctness,  
the outcome of such  
an execution should match the  
sequential execution.

Process X

$A = 1$

$B = 2$

$A = B + 1$

$B = B * 2$

Sequential code  
fragment

one of  
the memory  
fences in  
this scenario

Process Y

$X1.$

$A = 1$

$Y1.$

$B = 2$

$\Rightarrow X2. A = B + 1$

$Y2. B = B * 2$

“Memory fence”  
or “barrier”

that enforces an  
operation order

Results of X2 execution

Should be committed  
to memory before  
Y2 is executed.

# Race Conditions

- When two or more processes are reading or writing some shared data and final result depends on who runs precisely when is a *race condition*
- How to avoid race conditions?
  - ◆ Prohibit more than one process from reading and writing shared data at the same time
  - ◆ Essentially we need *mutual exclusion*
- Mutual exclusion:
  - ◆ When one process is reading or writing a shared data, other processes should be prevented from doing the same
  - ◆ Providing mutual exclusion in OS is a major design issue

See the power point slides for the animated version of this slide.

## An Example

Correct balance = \$100  
(\$50 withdrawal and \$50 deposit)

Initial balance = \$100

### Process A

R1 <- \$100

R2 <- \$50

R1 <- \$100 - \$50

balance <- R1 = \$50  
STORE R1, balance

A & B are updating a  
balance. A is withdrawing and B is depositing.

### Process B

R3 <- \$100

R4 <- \$50

R3 <- \$100 + \$50

balance <- R3 = \$150  
STORE R3, balance

Possible values for balance are \$50, \$150, and \$100!

Banking Problem - we have two processes or threads that are modifying a **shared variable** - the **account balance**

Withdraw Process

// Retrieve balance in R1

$R1 \leftarrow \$100$  (balance)

// subtract the amount

$R1 \leftarrow R1 - \$50$

// update the balance

$R1 \rightarrow \text{balance}$

Deposit Process

// Retrieve balance in R1

$R1 \leftarrow \$100$  (balance)

// Add the dep. amount

$R1 \leftarrow R1 + \$50$

// update the balance

$R1 \rightarrow \text{balance}$

How could we end up with \$150 and \$50 due to race conditions?

The following execution sequence gives \$50.

$R1 \leftarrow \$100$  (balance)

$R1 \leftarrow R1 - \$50$

← we switch from withdraw to deposit before withdraw was able to commit its computed R1

$R1 \leftarrow R1 + \$50$

$R1 \rightarrow$  balance  $\leftarrow \$150$  in balance.

$R1 \rightarrow$  balance

$\leftarrow$  \$50 is written into balance by the resumed deposit process.

Solution: To perform the balance update in a critical section.

# An Example....

- Suppose we have the following code for the account transactions

authenticate user

open account

load R1, balance

load R2, amount (-ve for withdrawal)

add R1, R2

store R1, balance

close account

display account info

# An Example....

- Suppose we have the following code for the account transactions

authenticate user

open account

load R1, balance

load R2, amount (-ve for withdrawal)

add R1, R2

store R1, balance

close account

display account info

Critical section

Question: Why would the introduction of critical section solve the race condition introduced by the "balance update" problem?

Critical section use prevents an instruction interleaving like the one shown before that causes a race condition. It does not allow another thread to start executing a statement in the CS while a thread is still executing one of it.

This prevents problematic interleaving of statements.

# Critical Section

- Part of the program that accesses shared data
- If we arrange the program runs such that no two ~~programs~~ processes are in their critical sections at the same time, race conditions can be avoided

# Critical Section

- For multiple programs to cooperate correctly and efficiently:
  - ◆ No two processes may be **simultaneously** in their critical sections
  - ◆ No assumptions be made about **speeds** or number of CPUs
  - ◆ No process running **outside** its critical section may block other processes
  - ◆ No process should have to **wait forever** to enter its critical section

# Critical Sections

- Critical region execution should be *atomic* i.e., it runs “all or none”
- Should not be interrupted in the middle
- For efficiency, we need to minimize the length of the critical sections
- Most inefficient scenario
  - ◆ Whole program is a critical section – no multiprogramming!

# Road to a Solution

- Simplest solution:
  - ❖ Disable all interrupts
  - ❖ This should work in a single processor situation
  - ❖ Because interrupts cause out-of-order execution due to interrupt servicing
  - ❖ With interrupts disabled, a process will run until it yields the resource voluntarily
- Not practical:
  - ❖ OS won't allow a user process to disable all interrupts – OS operation will be hindered too!
  - ❖ Does not work on multiprocessors

# Road to a Solution

- Idea: To come up with a software solution
- Use “lock” variables to prevent two processes entering the critical section at the same time – all along we are talking about a single critical section
- Use variable `lock`
- If `lock == 0` set `lock = 1` and `enter_region`
- If `lock == 1` wait until `lock` becomes 0
- Does not work, Why??

var lock = 0

while (lock);  
lock = 1;

Critical  
Section

lock = 0

entry code

lock taking

exit code

lock releasing

lock = 0

while (lock); empty statement

====

lock = 0

The simple idea of using a “lock” variable does not work due to the same problem we encountered with the bank balance update problem.

# Strict Alternation

- Two processes take turns in entering the critical section
- Global variable turn set either to 0 or 1

Process 0

```
while (TRUE) {  
    while (turn !=0);  
    critical_section();  
    turn = 1;  
    non_critical();  
}  
}
```

Process 1

```
while (TRUE) {  
    while (turn !=1);  
    critical_section();  
    turn = 0;  
    non_critical();  
}
```

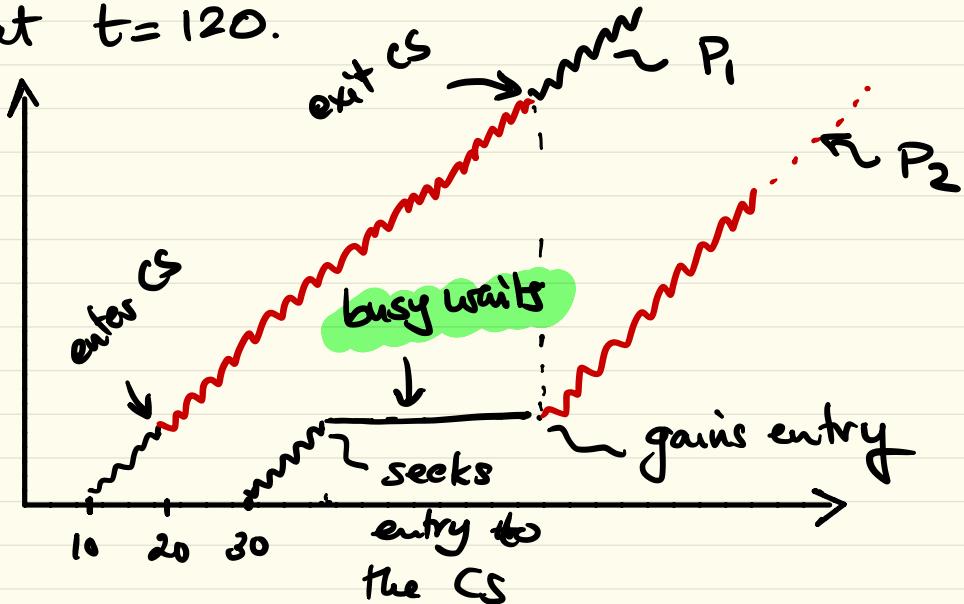
# Strict Alternation

- What is the problem with strict alternation?
- We can have starvation, why??
- What are the other drawbacks?
  - ◆ Continuously testing a variable until some value appears is called **busy waiting**
  - ◆ Busy waiting wastes CPU time – should be used when the expected wait is short
  - ◆ A lock that uses busy waiting is called **spin lock**

Lets digress a bit. Suppose we assume that we have got a solution for the critical section problem that uses "busy waiting". We want to understand the pros and cons of this approach.

As the name implies with busy waiting the processes that are unable to access the critical section because another process is currently in it will wait for their turn while continuing to consume CPU cycles. This can cause slowdowns or even deadlocks.

Consider a simple scenario, where process  $P_1$  is created at  $t = 10$ . It enters the critical section at  $t = 20$  and leaves the CS at  $t = 120$ .



$P_2$  was busy waiting — running on the CPU and failing to grab the lock needed to enter the CS.

Because  $P_1$  and  $P_2$  are both running on the CPU, with a fair scheduling algorithm that evenly splits the CPU resources among the competing processes,  $P_1$  and  $P_2$  get half the CPU each — assuming no other process is in the system.

Example: We have a process that is running a task that takes 10s.

For 4s (first 4s), the task is in a CS.

Suppose we are running two instances of the task in two different processes, what is the exec. time?

Let's make the following assumptions:

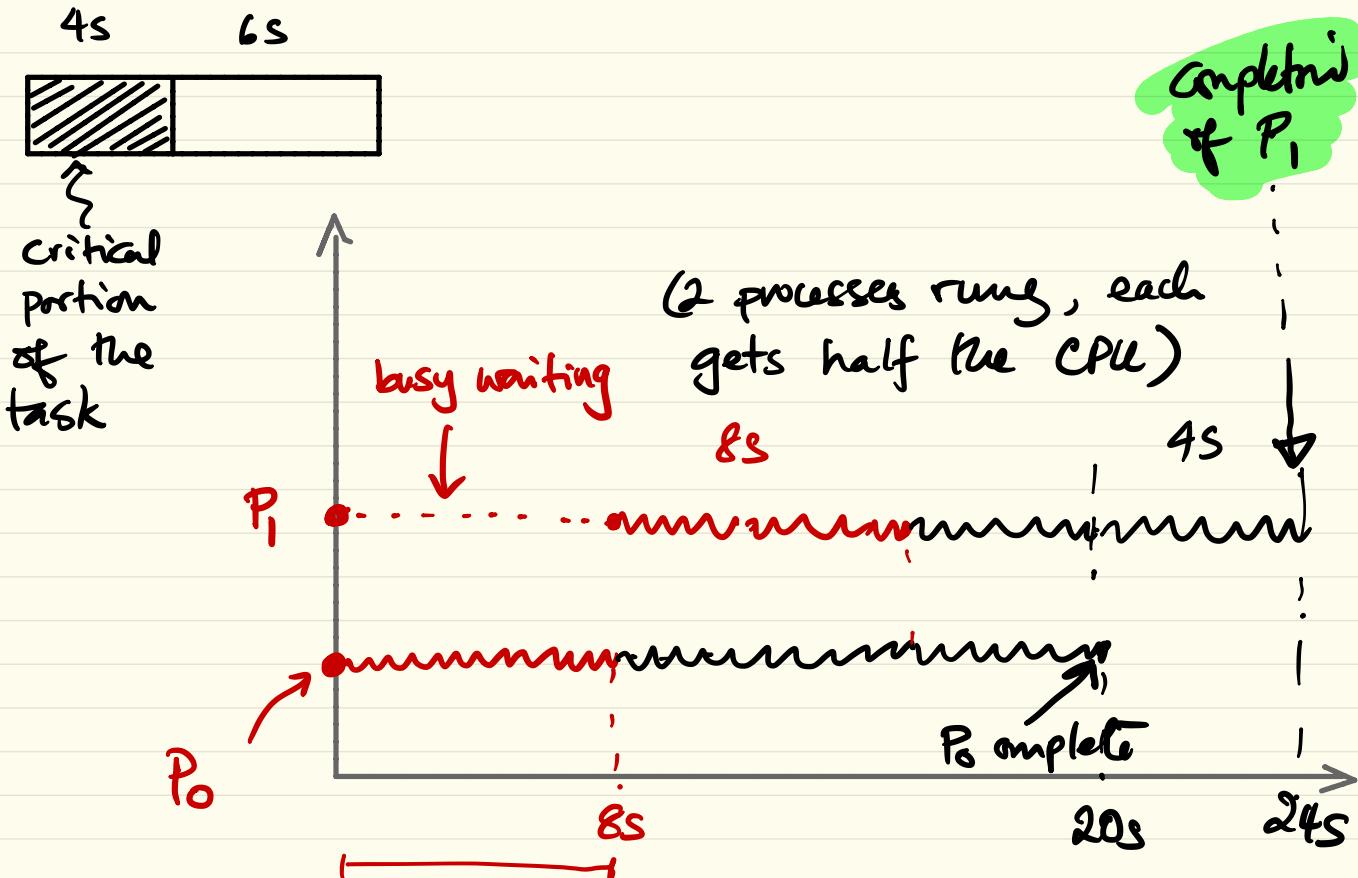
- single CPU, single core.
- Ideal time sharing, no other tasks besides these two tasks.

Simple argument: 2 tasks 20s (total workload)

So we need at least 20 seconds.

Due to busy waiting, we have wasted cycles.

One task gets into the CS, and for 4s the next task will be busy waiting. So the second task will end up with a 14s workload. So, we need 24s to complete both tasks.



It took 8s to process the CS of  $P_0$  because  $P_1$  was busy waiting

Completion of  $P_1$

Question: Suppose the length of the CS is 1.0s, what is the total run time?

What can you say about the use of busy waiting? Would you recommend its use in certain situations?

# Example

Kernel level multi-threads are used for concurrent processing in a data-parallel application. The application has a critical section. If a single thread of execution is used the critical section takes 2 seconds and other parts take 6 seconds. We have a large dataset so we use 2 threads. What is the run time? We have an even larger dataset so we use 8 threads. What is the run time?

Assume single CPU and ideal time sharing (no overhead). All threads starting at the same time.

# Locks – An Illustration



See the animation in the powerpoint  
slides

According to a book on Operating Systems,  
this is the first documented use of  
critical sections.

The "space" at the  
intersection of the  
propeller path and  
bullet path is the  
CS that needs to  
be protected so  
that only one is  
crossing it at any given time!



# Mutual Exclusion: First Attempt

- The simplest mutual exclusion strategy is taking turns as considered earlier

```
/* process 0 */  
.  
. .  
while (turn != 0);  
/* critical section */  
turn = 1;  
. .
```

```
/* process 1 */  
.  
. .  
while (turn != 1);  
/* critical section */  
turn = 0;  
. .
```

# Mutual Exclusion: Second Attempt

- First attempt problem – single key shared by the two processes
- Each have their own key to the critical section
- Solution does not work! Why??

```
/* process 0 */  
.  
.  
while (flag[1]);  
flag[0] = true;  
/* critical section */  
flag[0] = false;  
.  
.
```

testing  
problem:  
we are  
unable  
to test &  
set without  
interruption.

```
/* process 1 */  
.  
while (flag[0]);  
flag[1] = true;  
/* critical section */  
flag[1] = false;  
.
```

testing  
setting

We could have a context switch in 

# Mutual Exclusion: Third Attempt

- This works, i.e., provides mutual exclusion
- Has **deadlock** – why?

In the previous scenario, we have no CS due to context switches in between Testing & Setting

```
/* process 0 */  
.  
.  
flag[0] = true;  
while (flag[1]);  
/* critical section */  
flag[0] = false;  
.  
.
```

```
/* process 1 */  
.  
.  
flag[1] = true;  
while (flag[0]);  
/* critical section */  
flag[1] = false;  
.  
.
```

Here, we switch the order, we can get deadlock! No process moves forward.

# Mutual Exclusion: Fourth Attempt

- This works
- Has livelock – why?

```
/* process 0 */  
.  
. .  
flag[0] = true;  
while (flag[1])  
{  
    flag[0] = false;  
    /* random delay */  
    flag[0] = true;  
}  
/* critical section */  
flag[0] = false;  
.  
. .
```

```
/* process 1 */  
.  
. .  
flag[1] = true;  
while (flag[0])  
{  
    flag[1] = false;  
    /* random delay */  
    flag[1] = true;  
}  
/* critical section */  
flag[1] = false;  
.  
. .
```

One way of attempting to solve deadlock is by “being nice”. Where we give up

the demand and try to re-grab.

This can lead to livelock if both parties re-grab at the same instance.

# Dekker's Algorithm

```
/* process 0 */  
...  
flag[0] = true;  
while (flag[1])  
{  
    if (turn == 1)  
    {  
        flag[0] = false;  
        while (turn == 1);  
        flag[0] = true;  
    }  
}  
/* critical section */  
turn = 1;  
flag[0] = false;  
...  
...
```

```
/* process 1 */  
...  
flag[1] = true;  
while (flag[0])  
{  
    if (turn == 0)  
    {  
        flag[1] = false;  
        while (turn == 0);  
        flag[1] = true;  
    }  
}  
/* critical section */  
turn = 0;  
flag[1] = false;  
...  
...
```

# Peterson's Algorithm

- Dekker's algorithm is complicated
  - ◆ hard to prove the correctness
- Peterson's algorithm is much simpler
- Based on the same idea of using the *turn* variable to arbitrate and an array of flags to express interest to enter the critical section

# Peterson's Algorithm

```
/* process 0 */  
...  
flag[0] = true;  
turn = 1;  
while (flag[1] &&  
        turn == 1);  
/* critical section */  
flag[0] = false;  
/* remainder */  
...
```

```
/* process 1 */  
...  
flag[1] = true;  
turn = 0;  
while (flag[0] &&  
        turn == 0);  
/* critical section */  
flag[1] = false;  
/* remainder */  
...
```

# Can Hardware Help?

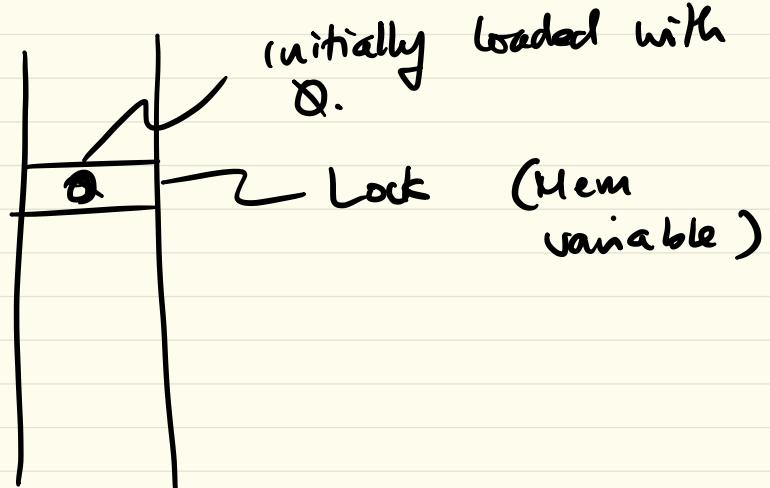
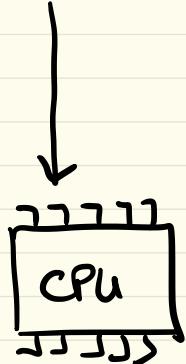
- Dekker's and Peterson's algorithms are pure software solutions
- Can hardware provide any help?
  - ◆ make the solution more efficient
  - ◆ make it scalable to more processes?
- Yes!
- Current microprocessors have hardware instructions supporting mutual exclusion

Recall from a earlier discussion that a simple idea of using a "lock" variable, could have worked for providing a critical section if we could do the **Testing** and **Setting** atomically. That is once the value is tested and ascertained to contain a certain value we perform a certain modification on the value, without interruption.

This is the basis of hardware implementations. There are various forms with subtle variations in their capabilities.

Here, we consider Test & Lock ( $R$ ,  $Lock$ )

TRL ( $R$ ,  $Lock$ )



$P_0$        $P_1$        $P_2$

Suppose  $P_0$ ,  $P_1$ , and  $P_2$  execute TRL ( $R$ ,  $L$ ) simultaneously,

we claim only one process will get 0 in its register.

# Test and Lock

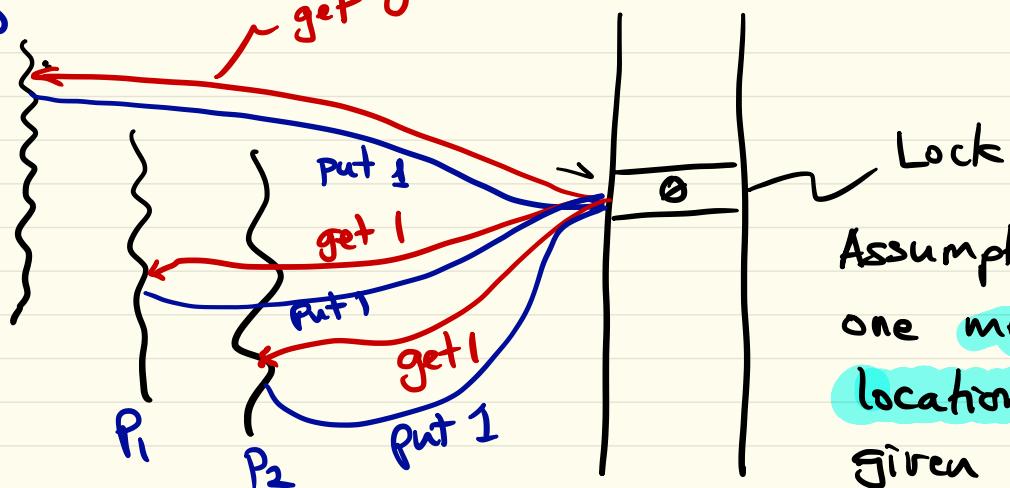
- TSL RX, LOCK is a typical CPU instruction providing support for mutual exclusion
  - ◆ read the contents of memory location LOCK into register RX and stores a non-zero value at memory location LOCK
  - ◆ operation of reading and writing are indivisible - atomic

enter\_section:

```
TSL REGISTER, LOCK
    // copy lock to reg and set lock to 1
    CMP REGISTER, #0
    // was lock zero??
    JNE enter_section
    // if non zero, lock was set
    RET
```

leave\_region:

```
MOVE LOCK, #0
RET
```



Assumption: Only one memory location for the given value.

$P_0$  gets the 0.  
 It gets the chance to get into the CS. Others wait (busy wait) -

The location - Lock - was initialized to 0. All processes are trying to grab this 0 and store a non-zero (1) value in there. Only the process will succeed in getting the 0.

# Properties of Machine Instruction Approach

## ⌘ Advantages:

- ❖ applicable to any number of processes
- ❖ can be used with single processor or multiple processors that **share a single memory**
- ❖ simple and easy to verify
- ❖ can be used to support multiple critical sections, i.e., define a separate variable for each critical section

# Properties of Machine Instruction Approach

## Disadvantages:

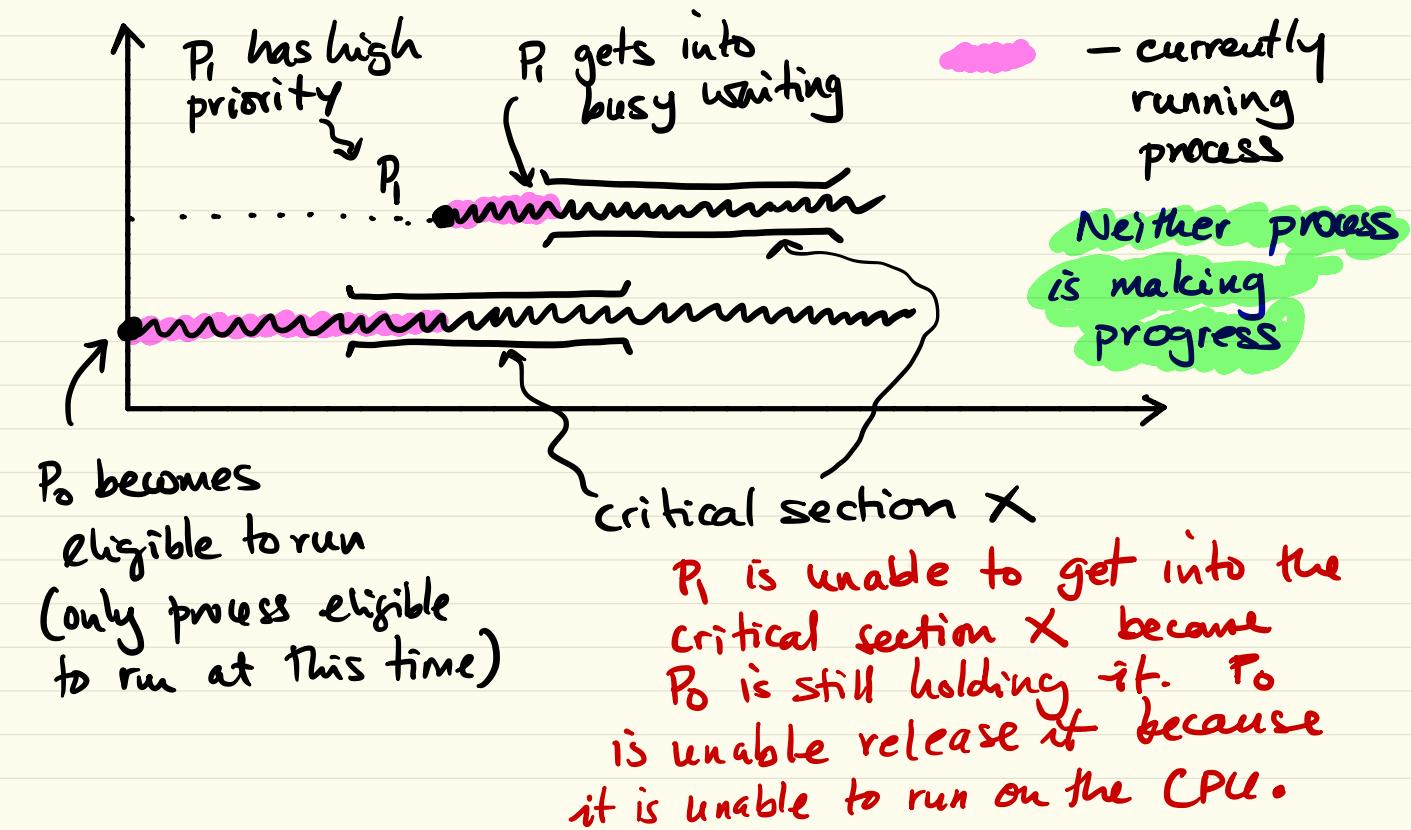
- ◆ **Busy waiting is employed** – process waiting to get into a critical section consumes CPU time
- ◆ **Starvation is possible** – selection of entering process is arbitrary when multiple processes are contending to enter

**Starvation** happens when a process is indefinitely delayed in getting its resources - because the system fails to choose it from a competing set of processes

# Busy Waiting Approaches

- Mutual exclusion schemes discussed so far are based on busy waiting
- Busy waiting not desirable:
- Suppose a computer runs two processes H: high priority and L: low priority
  - ❖ scheduler always runs H when it is in ready state
  - ❖ at a certain time L is in its critical section and H become runnable
  - ❖ H begins busy waiting to enter the critical section
  - ❖ L is never scheduled to leave the critical section
  - ❖ there is a **deadlock**
  - ❖ this situation is sometimes referred to as the **priority inversion problem**

To illustrate priority inversion, we use the following scenario.

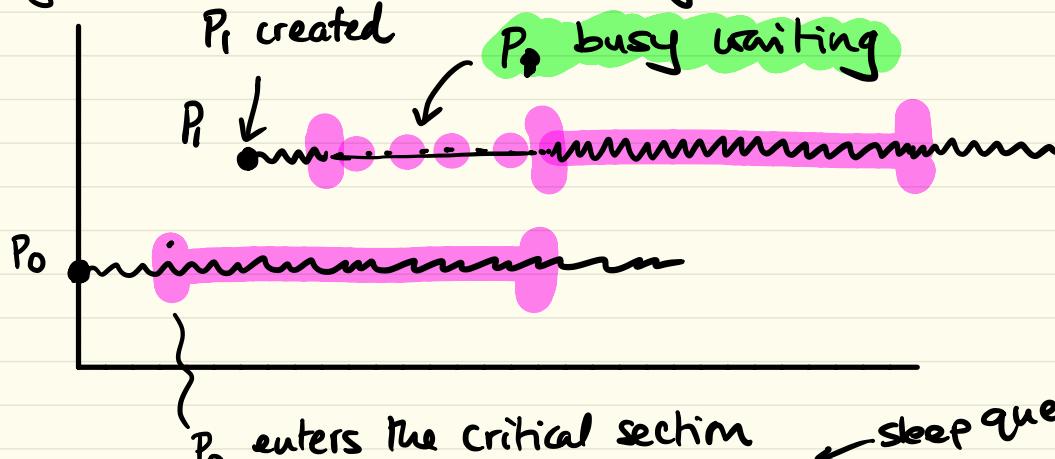


# Sleep/Wakeup Approach

- Alternative to busy waiting that is inefficient and deadlock prone is to use a sleep/wakeup approach
- Implemented by the Semaphores

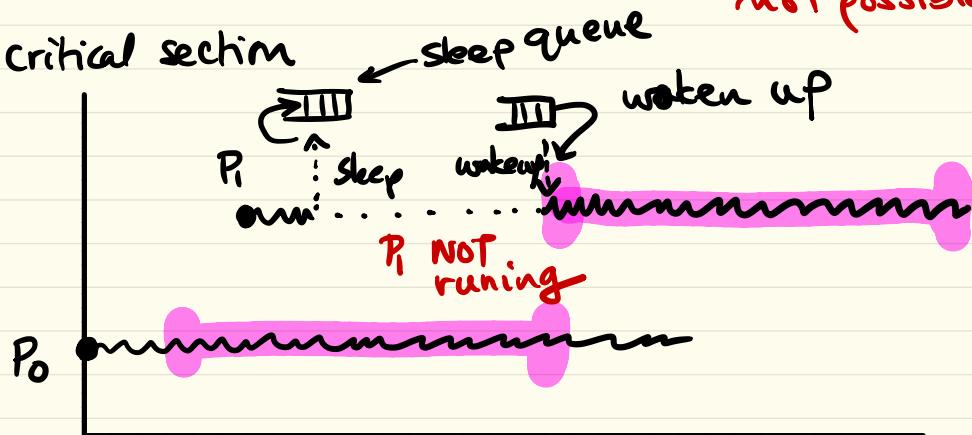
OS support is needed to implement process “sleep” and “wakeup”. Most OSes provide semaphores as part of their system call offering.

Let's contrast busy waiting and sleep/wakeup to get a better understanding.



There is overhead associated with putting a process to sleep and waking it up.

If the critical section is tiny, busy waiting could be a better alternative provided priority inversion is not possible.



# Semaphores

- Fundamental principle:
  - ◆ two or more processes can cooperate by sending simple messages
  - ◆ a process can be forced to stop at a specific place until it receives a specific message
  - ◆ complex coordination can be satisfied by appropriately structuring these messages
  - ◆ for messaging a special variable called semaphore **s** is used
  - ◆ to transmit a message via a semaphore a process executes signal(s)
  - ◆ to receive a message via a semaphore a process executes wait(s)

# Semaphores

- ⌘ Operations defined on a semaphore:
  - ◆ can be initialized to a **nonnegative value** – set semaphore
  - ◆ wait – decrements the semaphore value – if value becomes negative, process executing wait is blocked
  - ◆ signal – increments the semaphore value – if **value is not positive**, a process blocked by a wait operation is unblocked

# Semaphores

- A definition of semaphore primitives...

```
struct semaphore {  
    int count;  
    queueType queue;  
}
```

```
void wait(semaphore s)  
{  
    s.count--;  
    if (s.count < 0)  
    {  
        place this process in  
        s.queue;  
        block this process  
    }  
}
```

```
void signal(semaphore s)  
{  
    s.count++;  
    if (s.count <= 0)  
    {  
        remove a process P  
        from s.queue;  
        place process P on  
        ready list;  
    }  
}
```

# Semaphores

- Wait and Signal primitives are assumed to be atomic
  - ◆ they cannot be interrupted and treated as an indivisible step
- A queue is used to hold the processes waiting on a semaphore
- How are the processes removed from this queue?
  - ◆ FIFO: process blocked longest should be released next – ***strong semaphore***
  - ◆ Order not specified – ***weak semaphore***

# Type of semaphores

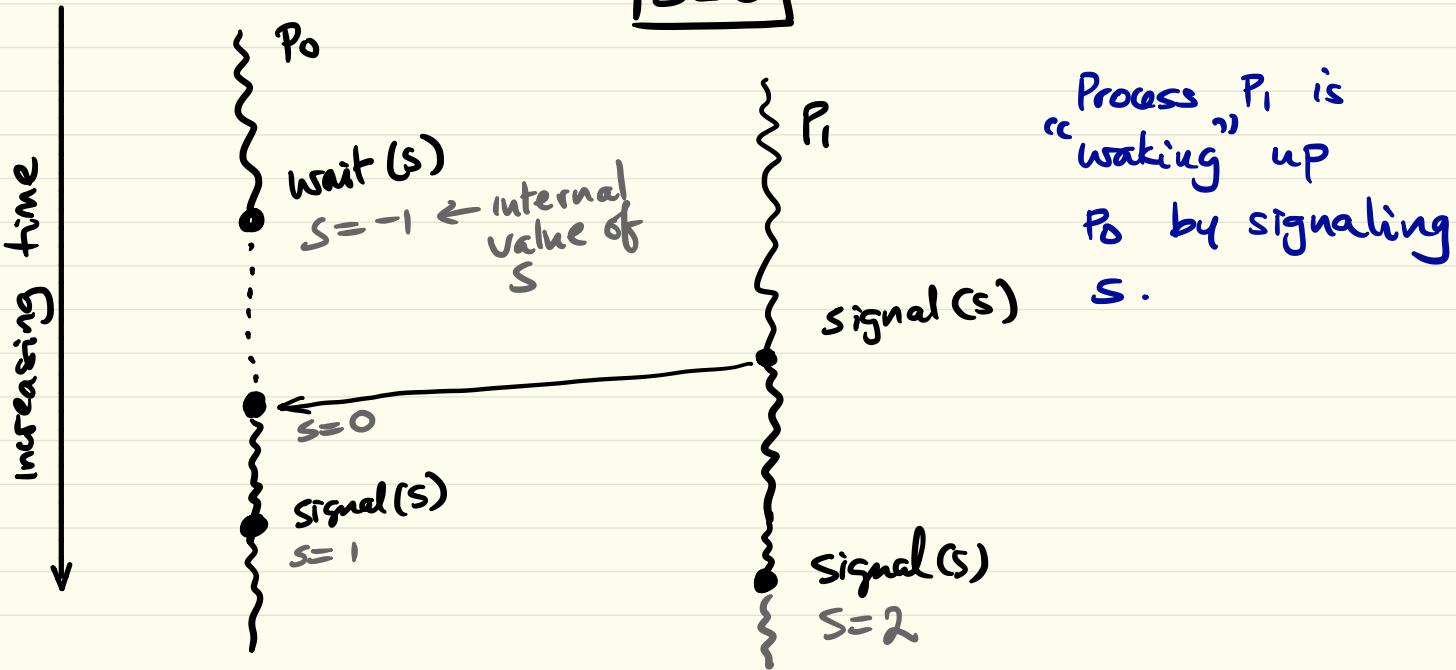
## ■ Semaphores are usually categorized into two ways:

- ◆ **binary**—is a semaphore with an initial value of 0 or 1
  - Or a value of FALSE or TRUE if you prefer
  - Initialized to 1 for mutex applications
- ◆ **counting**—is a semaphore with an integer value ranging between 0 and an arbitrarily large number - initial value might represent the number of units of the critical resources that are available - also known as a **general** semaphore

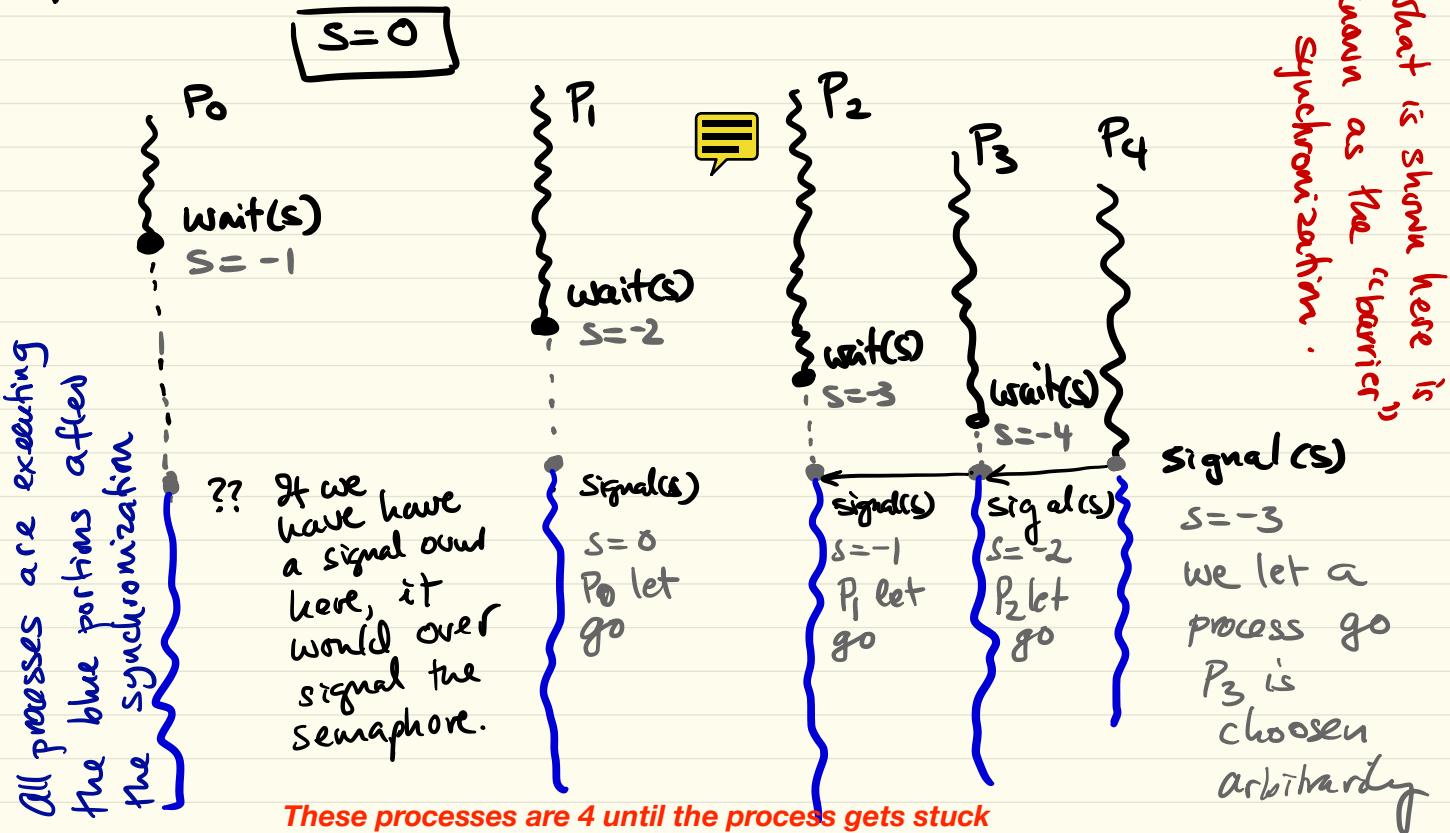
These are initial values.  
A semaphore can have negative value at any time .. many threads or processes waiting on it

To understand the operation of the semaphore operations lets consider a bunch of processes that share a single semaphore  $s$  that is initialized to 0.

$|s=0$  ← initialized.



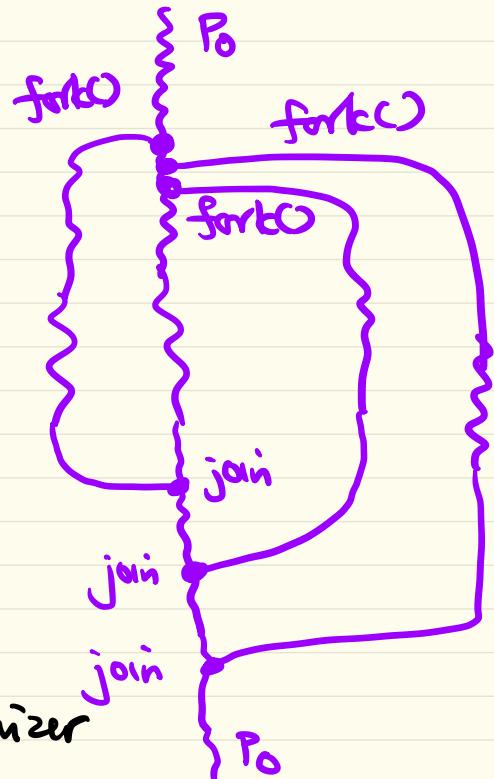
Let's consider another scenario. Here we have 5 processes. We want to synchronize their executions.



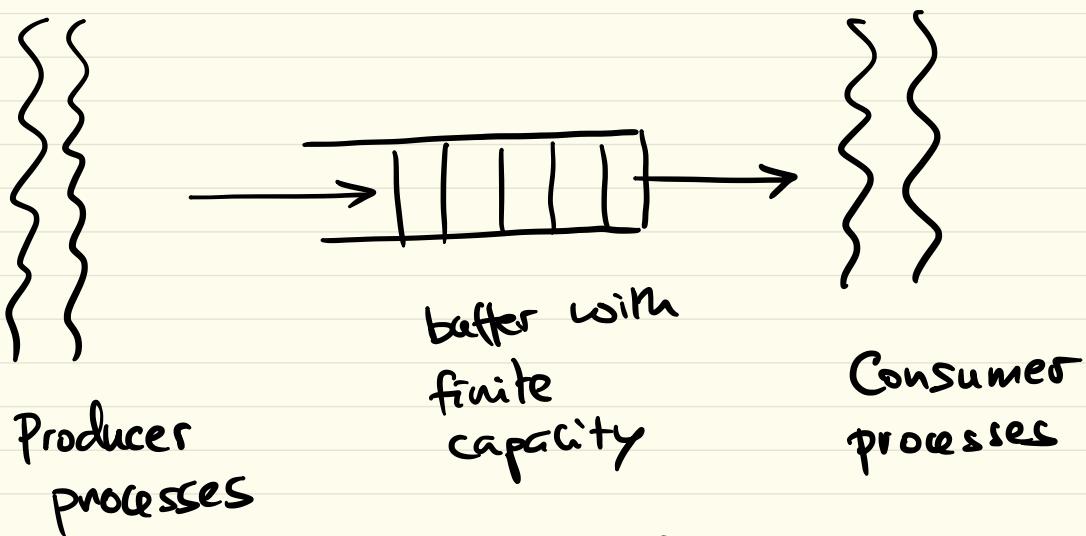
As another example consider a **fork-join graph** like what is shown next.

The “join” is a synchronization between two different processes. The process that arrives first at the join point needs to wait for the other process until its arrival.

So we can implement a join using a “barrier” type synchronizer for two processes.



# Producer - consumer Problem . (Bounded Buffer Version)



- ① Buffer can overflow
- ② Buffer can underflow (trying to read from empty buffer)
- ③ Buffer modified by both producer & consumer

So we have three synchronization requirements.



— overflow synchronization. That is, we want to prevent overflow and the resulting loss of data.



— underflow synchronization. We want to stop underflow. Otherwise, we will read corrupted or duplicated data.



— mutual exclusion synchronization. We want only one process modifying the shared buffer structure for race-free operation.

# Producer-Consumer: Semaphores

M

```
semaphore mutex = 1;  
semaphore empty = N;  
semaphore full = 0;
```



```
producer() {  
    int item;  
  
    while(TRUE) {  
        item = produce_item();  
        wait(&empty);  
        wait(&mutex);  
        insert_item(item);  
        signal(&mutex);  
        signal(&full);  
    }  
}
```

Check



impl.

```
// protects the critical section  
// counts the empty slots  
// counts full buffer slots
```

```
consumer() {  
    int item;
```

```
    while(TRUE) {  
        wait(&full);  
        wait(&mutex);  
        item = remove_item();  
        signal(&mutex);  
        signal(&empty);  
        consume_item(item);  
    }  
}
```

check



# Producer-Consumer: Semaphores

- One binary semaphore **mutex** to ensure only one process is manipulating the buffers at a time
- **Empty** and **Full** counting semaphores, to count the number of empty or full buffer slots respectively
  - ◆ Empty initialized to  $N$ , so waiting on empty means waiting if there are no empty slots (*buffer full!*)
  - ◆ Full initialized to 0, so waiting on full means waiting if there are no full slots (*buffer empty!*)

# Primitives Revisited

- The synchronization primitives discussed so far are all of the form:

**entry protocol**

**< access data >**

**exit protocol**

- Semaphores give us some abstraction: implementation of the protocol to access shared data is transparent to the user
- More abstraction would be of additional benefit (as it often is!)

# Monitors

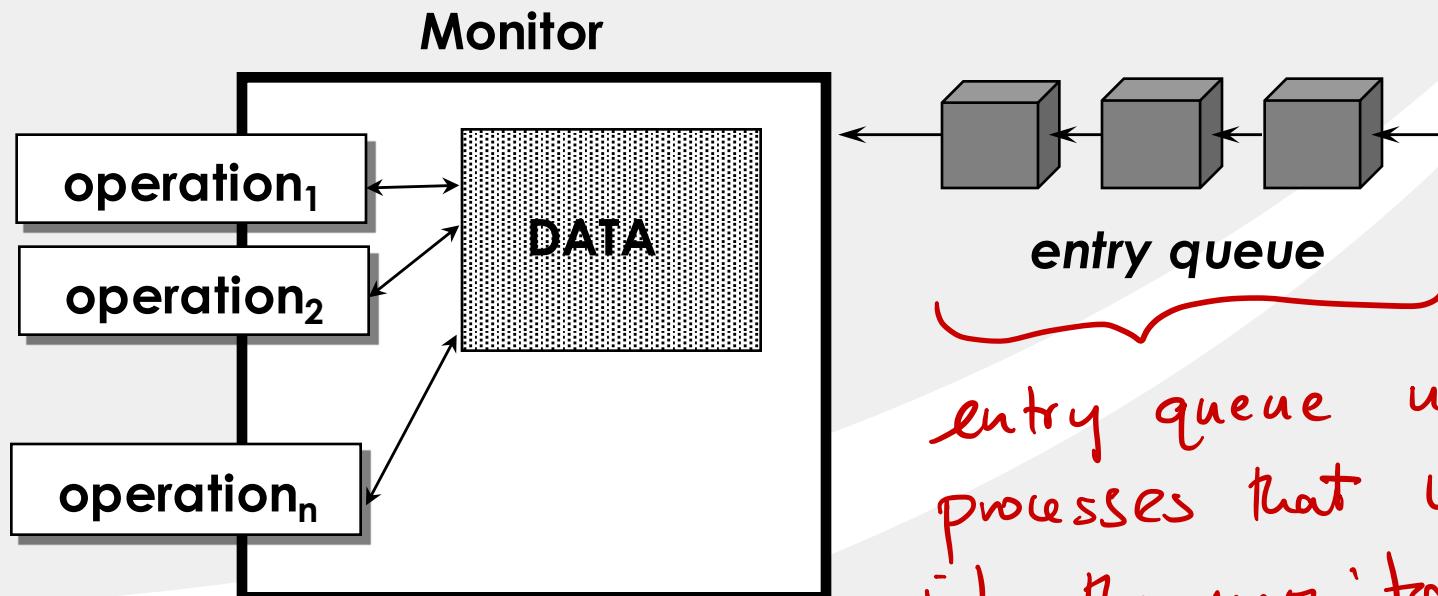
- A monitor is a high-level (programming language) abstraction that combines (and hides) the following:
  - shared data
  - operations on the data
  - synchronization using condition variables

# Monitors

- Mutual exclusion --- not the only thing we need for concurrency
- Abstraction provided by monitors allows us to deal with many different issues in concurrency
  - ◆ block processes when a resource is busy
- With a monitor data type, we define a set of variables that define the state of an instance of the type, and a set of procedures that define the externally available operations on the type

# Monitors

- Monitors are more than just objects
- A monitor ensures that only one process at a time can be active within the monitor – so you don't need to code this explicitly



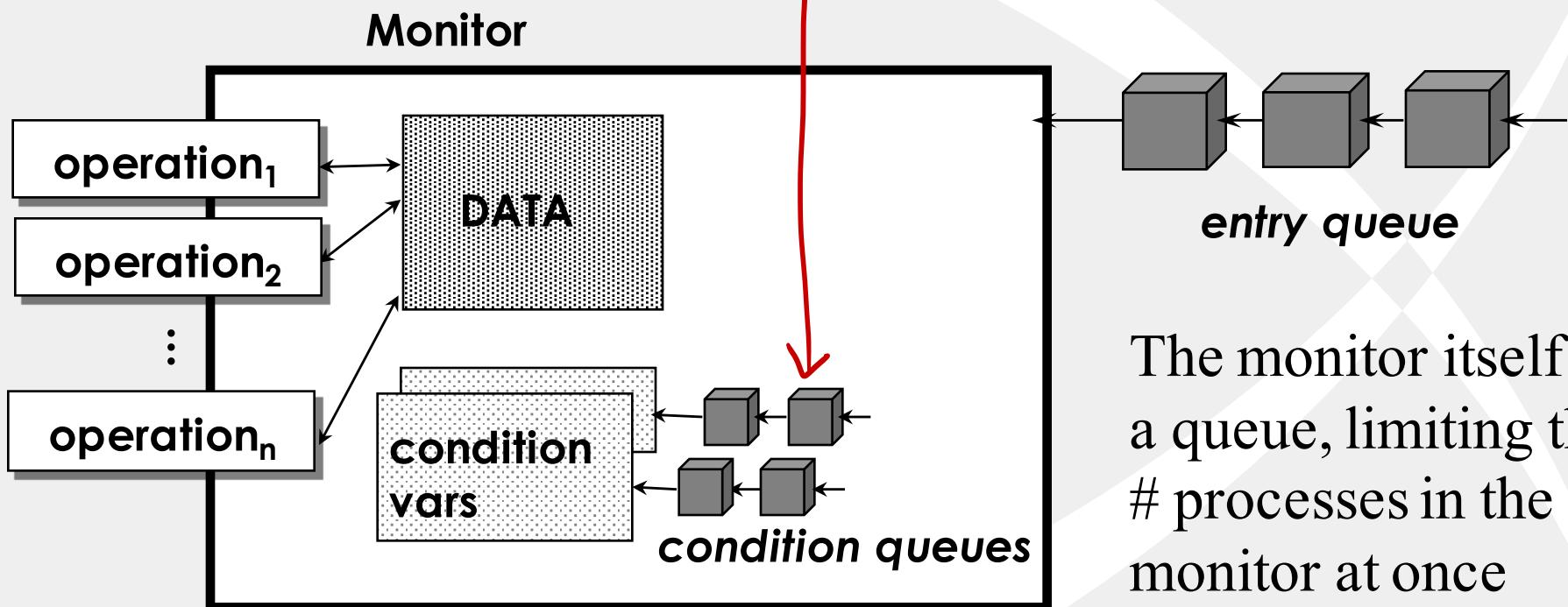
entry queue used by  
processes that want to get  
into the monitor but unable  
because the monitor is busy.

# Monitors

- Just this on its own though isn't enough – need the additional synchronization mechanisms
- Monitors use **condition variables** to provide user-tailored synchronization and manage each with a separate queue for each condition
- The only operations available on these variables are **WAIT** and **SIGNAL**
  - ◆ **wait** suspends process that executes it until someone else does a **signal** (different from semaphore wait!)
  - ◆ **signal** resumes one suspended process. no effect if nobody's waiting (different from semaphore free/signal)!

# Monitor Abstraction

Condition variables are used to halt processes within the monitor



The monitor itself has a queue, limiting the # processes in the monitor at once

A process gets "into" the monitor by running an operation that is exposed by the monitor.

# How Monitors Work

- Remember **only one process can be active** in the monitor at once. Some will be waiting to get in, others will be waiting on conditions in the monitor
- If **P executes `x.signal`** (signal for condition `x`), and there's a process `Q` suspended on `x`, **both can't be active at once**. 2 possibilities:
  - ◆ `P` waits on some condition till `Q` leaves monitor
  - ◆ `Q` waits on some condition till `P` leaves monitor
- Also compromises: if `P` executes `signal`, it leaves monitor automatically, resuming `Q`

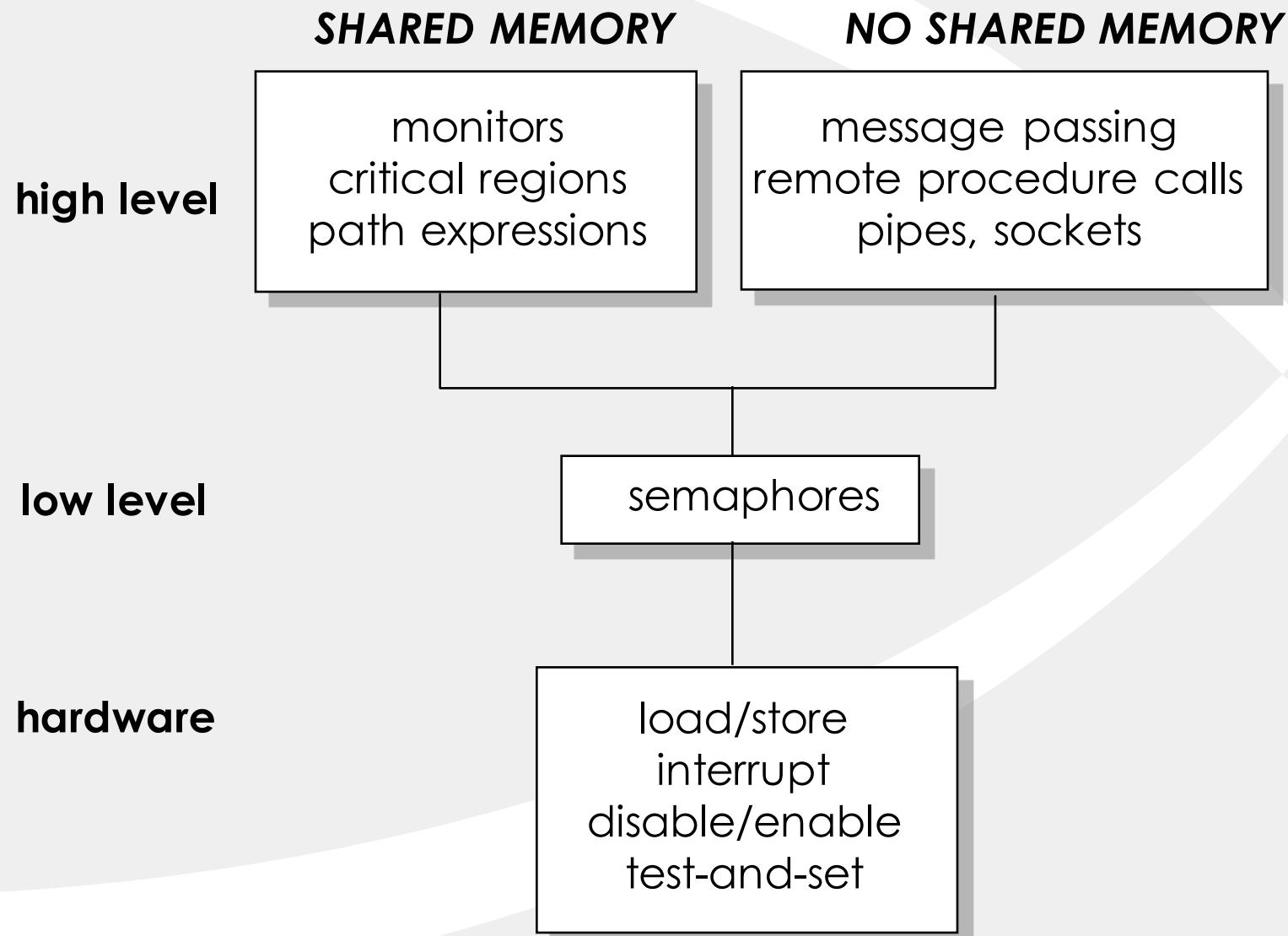
# Problems with *synch* Primitives

- *Starvation*: the situation in which some processes are making progress toward completion but some others are locked out of the resource(s)
- *Deadlock*: the situation in which two or more processes are locked out of the resource(s) that are held by each other

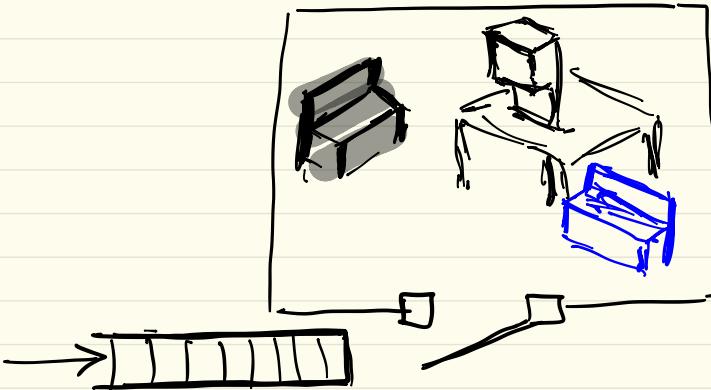
# Problems with *synch* Primitives

- The most important deficiency of the primitives discussed so far is that they were all designed on the concept of one or more CPUs accessing a ‘‘common’’ memory
  - ◆ Hence, these primitives are not applicable to distributed systems. **Solution?** Message passing...

# *Synch Primitives—Summary*



# Monitor Ville



## Ice creme maker

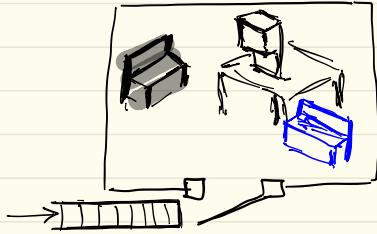
- enter the shop
- put ice creme
- \* if drinkers on black bench wake them
- leave the shop

## Icecreme drinker

- enter the shop \*
- get ice creme
- if no ice creme
- \* goto sleep on black bench
- if got ice creme
- \* leave the shop

Details are missing  
at \*

## Monitor Ville



### Ice cream maker

- enter the shop
- put ice cream
- \* if drinkers are black  
bench wake them
- leave the shop.

### Ice cream drinker

- enter the shop \*
- get ice cream
  - \* if no ice cream  
goto sleep on  
black bench
  - if got ice cream  
\* leave the shop

Details are missing  
at \*

### enter the shop:

- only one drinker  
or maker can  
be in the shop  
at any given time

- Sleeping drinkers  
or makers don't  
count.

### wake up a drinker:

- wake up me drinker and  
immediately goto sleep on  
the blue bench.

goto sleep!

be nice()

sleep on the black bench

### leave:

be nice();

leave shop;

### be nice:

if someone on blue  
bench wakes  
that person

else

let a new person  
into the shop

How about complex scenarios?

- Ice creme maker never sleeps on the black bench.
- what modification to the shop would require a black bench for the ice creme maker?

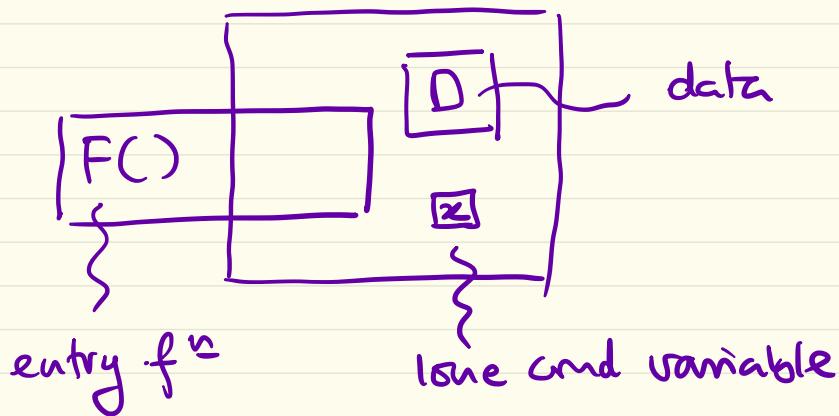
Go into the ice creme shop, you get ice creme and ketchup?

- First ice creme - always.
- Second ketchup - optionally - some drinkers want both some just ice creme
- What modifications?

After giving away ice cream for free for a long time, the ice cream maker closed shop. However, the shop was reused by the city to setup a water fountain.

- Still the same rules apply
- Only one drinker (water) in the shop at any given time. — always finds water
- No need for a black bench. — never run out of water!
- The blue bench comes with the shop and remains unused.

# Implement a monitor using Semaphores



Need three Semaphores : mutex = 1

x-sem = 0

next = 0

Some count variables : { next-count = 0  
x-count = 0 }

(ints)

m.F();

enter the  
monitor.

x.signal()

do signal while  
inside the monitor

x.wait()

go to sleep  
inside the monitor

m.F();

enter the monitor.

x.signal()

do signal while  
inside the monitor

x.wait()

go to sleep  
inside the monitor

wait (mutex);

F();

if (meat-cont > 0)  
signal (meat)

else

signal (mutex);

m.F();

enter the monitor.

x.signal()

do signal while  
inside the monitor

x.wait()

go to sleep  
inside the monitor

if (x-count > 0) {  
 next-count++;  
 signal (x-sem);  
 wait (next);  
 next-count--;  
}

m.F();

enter the monitor.

x.signal()

do signal while  
inside the monitor

x.wait()

go to sleep  
inside the monitor



x-cont++;  
if (next-cont > 0)  
    Signal(next)  
else  
    Signal(context));  
wait(x-sem);  
x-cont--;

We have seen two types of synchronization mechanisms

To show that they are equally expressive we show how one mechanism could be implemented using the other mechanism.

- ① Show semaphores can be simulated using monitors
- ② Monitors can be simulated using semaphores.

Let's look at the first simulation next. We already examined how the second simulation can be done in the preceding slides.

Monitor Semaphore {

condvar semwait;

int count;

signal() {

count++;

if (count <= 0)

semwait. signal();

}

wait() {

count--;

if (count < 0)

semwait. wait();

}

init (int val) {

count = val;

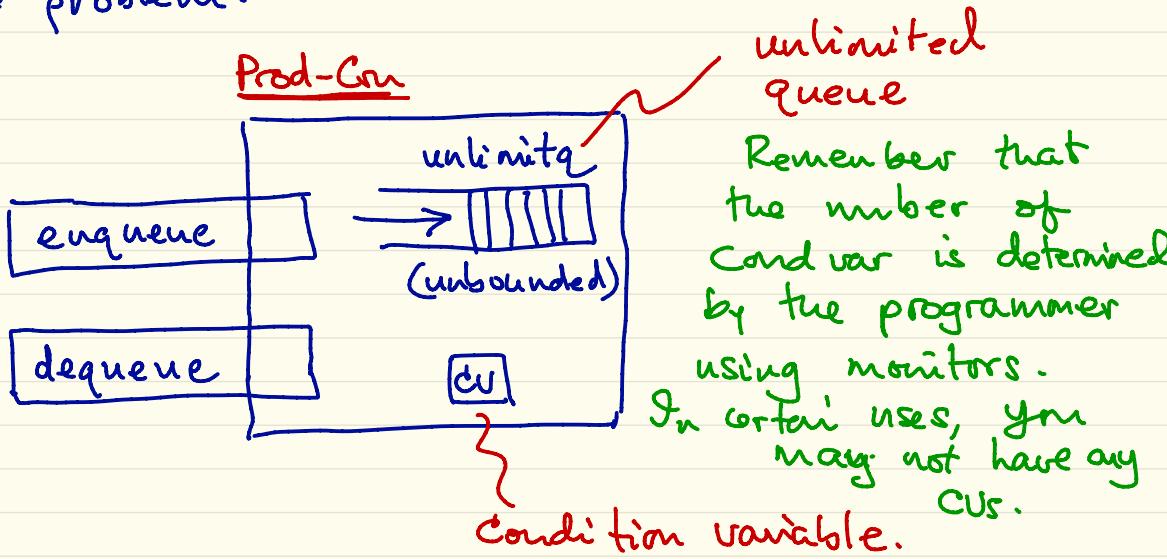
}

Implement an unbounded bounded buffer producer-consumer using monitors.

With unbounded buffer, we don't have a buffer overflow problem. Just the **underflow** and the **shared buffer update problem** remain.

We need a condvar for the underflow problem and the shared buffer update problem is handled by the monitor.

So we are going to create a monitor to solve the problem.



The monitor we want to develop looks like the above. It has two methods and has one condition variable for underflow protection.

We are going to use such a monitor as follows:

Monitor Prod-Con pc;

Producer () {

while () {

produce() → item

pc.enqueue(item);

}

}

Consumer () {

while () {

pc.dequeue() → item

consume(item);

}

use of our  
monitor

}

use of our  
monitor

Now we need to implement the monitor to realize the intended operations.

Monitor Prod-Con {

condvar emptybuf;

queue unlimitq;

void enqueue (item) {

unlimitq.push (item);

emptybuf.signal();

}

Any process that may be sleeping are woken up - only one is woken up.

item dequeue () {

if (unlimitq.size() == 0)

emptybuf.wait();

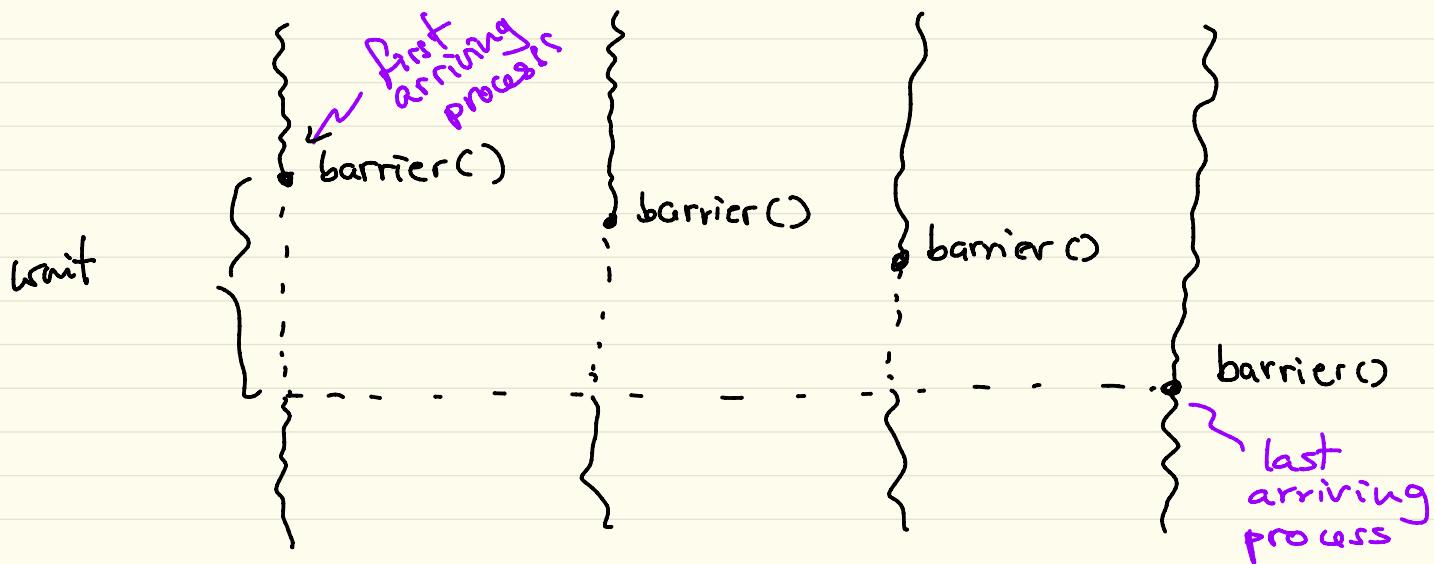
return unlimitq.pop();

F

A process trying to dequeue on an empty buffer goes to sleep here on the condvar emptybuf

EXERCISE: Implement a barrier() synchronization function for  $n$  processes using monitors.

Barrier() is a synchronization primitive where early arrivers are blocked until the late arrivers show up (call the barrier() function).



Your barrier() implementation can be of two different types:

- Reuseable implementation
- Non-reuseable implementation.

In the reuseable implementation, the barrier() will be in the initial state after all the synchronization of a barrier instance is complete. So it could be invoked again to perform another synchronization point.

**EXERCISE:** Implement a `qbarrier()` function with a quorum. In the barrier we saw previously, the barrier stopped all but the last process. The last process, unlocked all processes that arrived earlier.

With `barrier()`, we have computation phases, A and B, where we want all processes to have completed A before any one can start B. So we call the `barrier()` like the following :

`A();`

`barrier();`

`B();`

in each process that is engaged in the computing activity.

With the `qbarrier()`, we want to solve a slightly different problem.

We have  $n$  processes in the system. All of them are computing in phase `A()`. We can only enter phase `B()` after at least  $k$  processes have completed phase `A()`. The processes that arrive at the barrier the arrival of the  $k$ th process are not stopped at the barrier at all. So the barrier is open after the arrival of the first  $k$  processes.

`A();`  
`qbarrier (n, k);`  
`B();`

You could solve  
this problem  
using an existing  
barrier() implement.

**EXERCISE:** Implement a "prioritized synchronizer" using monitors.

The prioritized synchronizer psync object has two methods: pwait(), psignal().

A process can call pwait() with an argument like psync.pwait(5) to specify that it wants to wait at level 5. Levels with smaller numbers have higher priority. So Level 0 has the highest priority. Like the pwait(), the psignal() also has a priority.

psignal(5) will release a waiting process with the highest priority below Level 5. For example, if 3 processes are waiting at levels 3, 5, and 8.

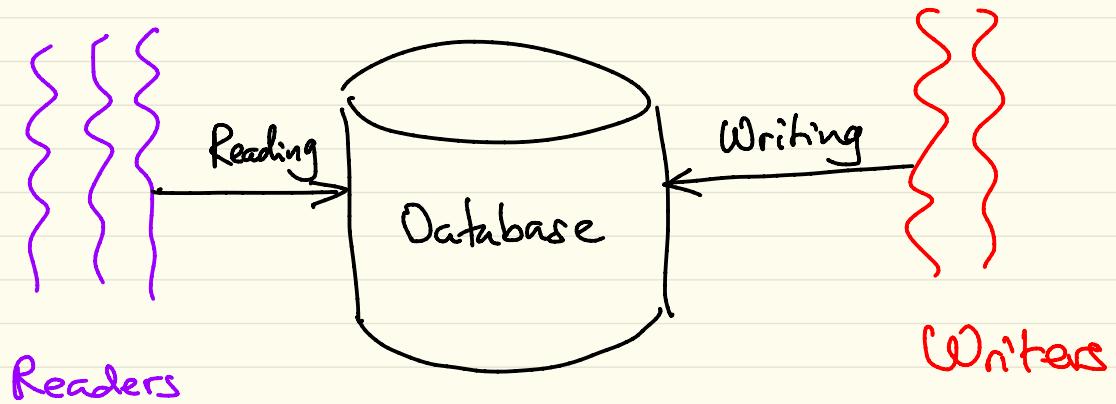
psignal(5) will release the process at level 6. So a signal at level 0 is bound to wake up a process if one is waiting. A signal at any other level can wake up no process because all of the waiting processes are at a higher level than the signalling level.

A signal has no other side effect besides waking up a process if one is waiting at the correct level.

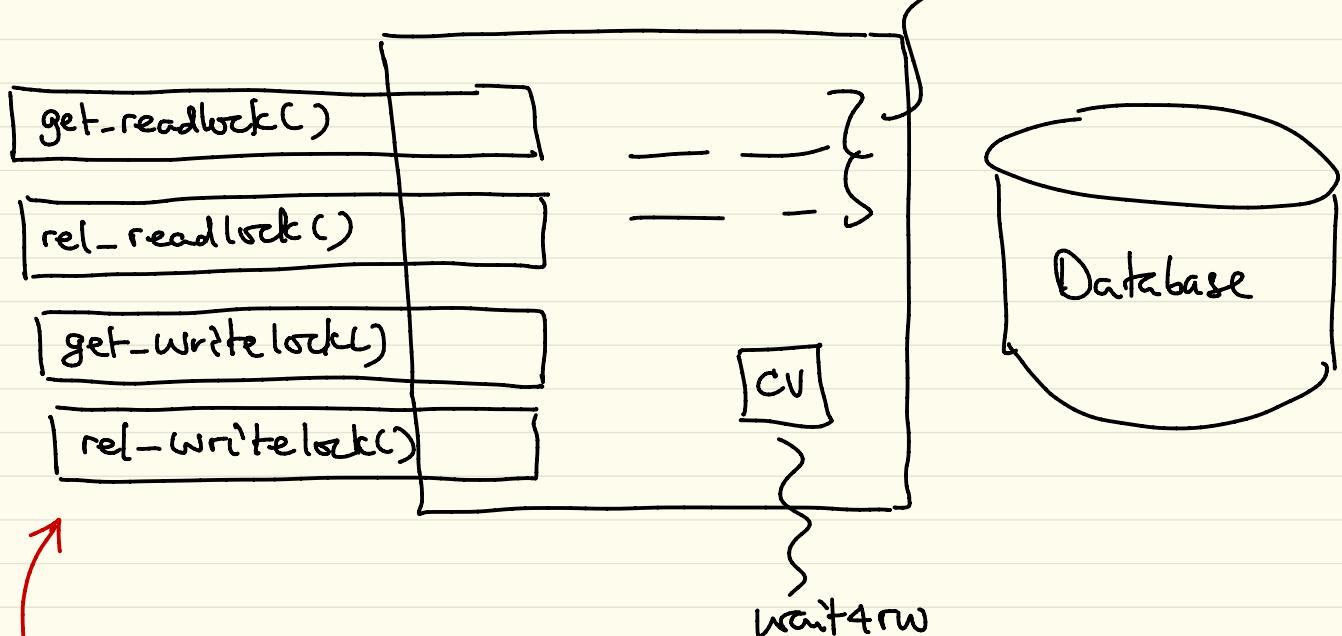
**EXERCISE:** Implement the readers-writers problem with readers having priority.

This is common interaction pattern that takes place in databases. In this pattern two or more readers can access the database at the same time.

However, a reader and writer cannot access at the same time. Either the reader or the writer have to wait. Same condition applies to two writers. No two writers can go at the same time. With the readers having priority, all readers who have arrived should complete before a writer is let to write. If there is a continuous inflow of readers, writers can starve.



# Monitor Readers-Writers



methods to be implemented  
as part of the Readers-Writers monitor.

variables.

Monitor Readers-Writers {

int readcnt;

bool writing, reading;

Condition wait4rw;

get-readlock();

}

release-readlock();

{

R

{ get-rlck();

Read DB

rd-lock();

init() {

readcnt = 0;

reading = false;

writing = false;

}

get-writelock();

{

release-writelock();

{

W

{ -get-wlck();

Write

{ -rel-wlck();

```
Monitor Readers-Writers {  
    int readcnt;  
    bool writing, reading;  
    condvar wait4rw;  
    get-readlock() {  
    }  
    release-readlock() {  
    }  
}
```

```
    get-writelock() {  
    }  
    release-writelock() {  
    }  
}
```

```
    rel-readlock() {  
        readcnt--;  
        if (readcnt == 0) {  
            reading = false;  
            wait4rw.wait();  
        }  
    }  
}
```

get-readlock() {  
 readcnt++;  
 if (reading)  
 return;  
 if (readcnt == 1  
 & !writing)  
 return;  
 wait4rw.wait();  
}

reading = true

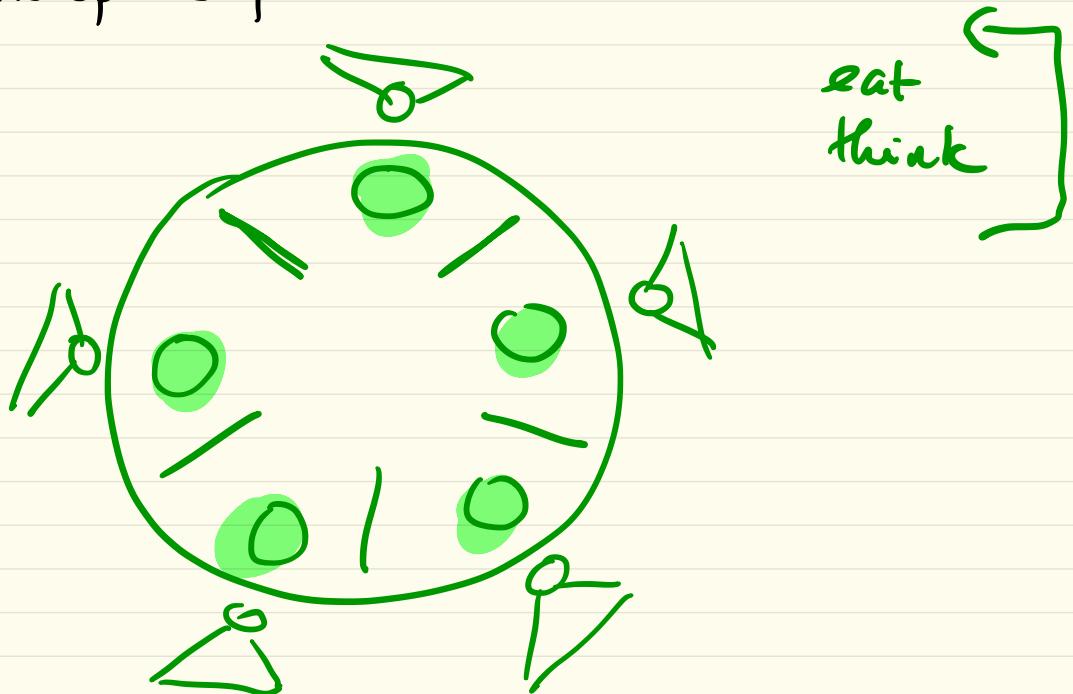
```
Monitor Readers-Writers {  
    int readcnt;  
    bool writing, reading;  
    condvar wait4rw;  
    get-readlock() {  
        {  
            release-readlock() {  
                {  
                    get-writelock() {  
                        {  
                            release-writelock() {  
                                {  
                            }  
                        }  
                    }  
                }  
            }  
        }  
    }  
}
```

```
get-writelock() {  
    if (reading || writing)  
        wait4rw-lock();  
    writing = true;  
    }  
}
```

```
rel-writelock() {  
    writing = false;  
    wait4rw.signal();  
}
```

3

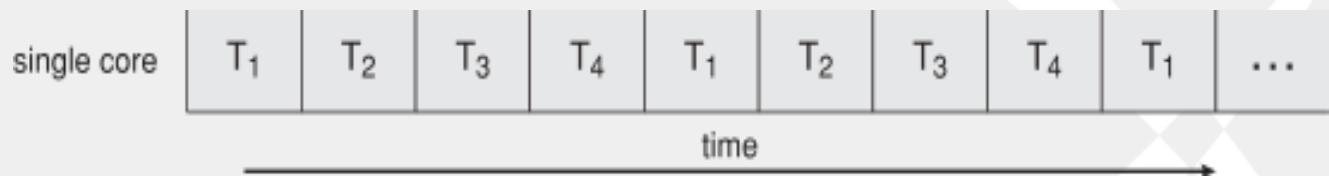
Implement a monitor to solve the  
dining philosophers' problem.



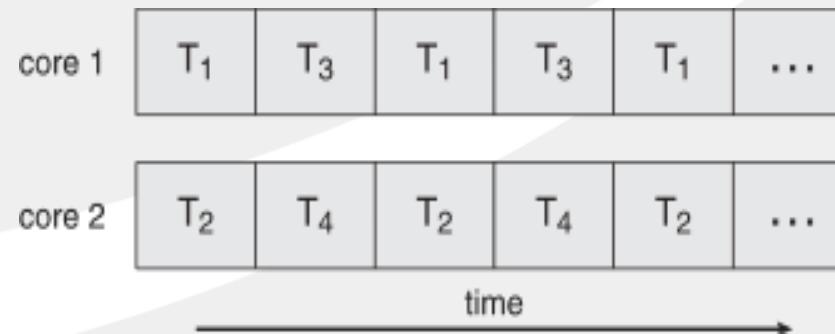
# Processes and Threads (3)

# Concurrency vs. Parallelism

- Concurrent execution on single-core system:

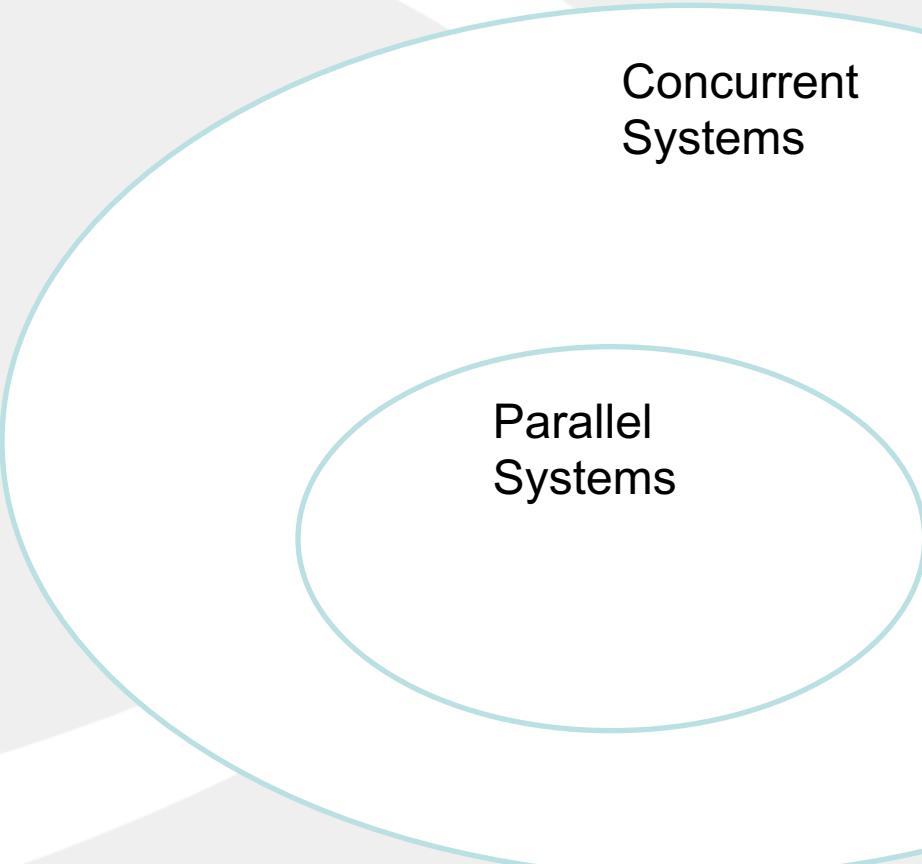


- Parallelism on a multi-core system:



# Concurrency vs. Parallelism

- **Parallelism** implies a system can perform more than one task simultaneously
- **Concurrency** supports more than one task making progress
  - ❖ Single processor / core, scheduler providing concurrency



Concurrent Systems

Parallel Systems

# Purpose of Parallelism

- Why do we want to use parallelism?
  - ◆ We want to run some computation faster
  - ◆ How fast can we go is an interesting question
- Can we keep on reducing the computation time

# Application Structure

- Applications are not completely parallel
  - ◆ Serial portion
  - ◆ Parallel portion

# Example

# Amdhal's Law

- Performance improvement obtained by applying an enhancement on an application execution is limited by the fraction of the time the enhancement can be applied.

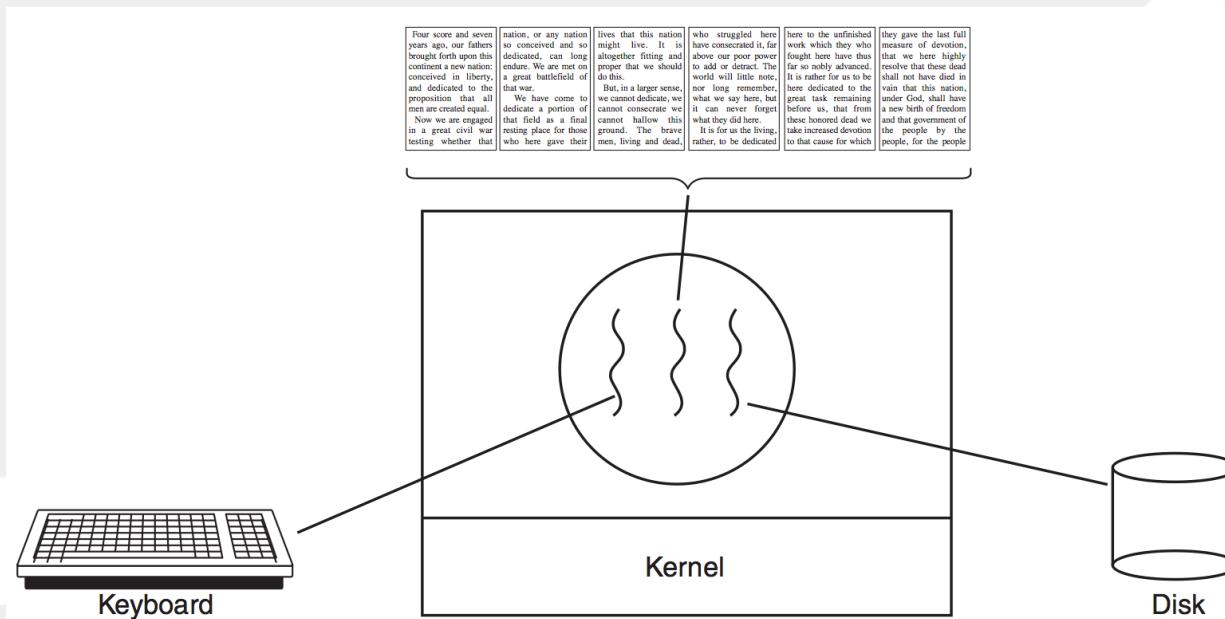
# Speedup for an Application

- Compute the speedup obtainable by adding  $N$  cores to an application
- $S$  is serial portion
- $1-S$  is the parallel portion

$$speedup \leq \frac{1}{S + \frac{(1-S)}{N}}$$

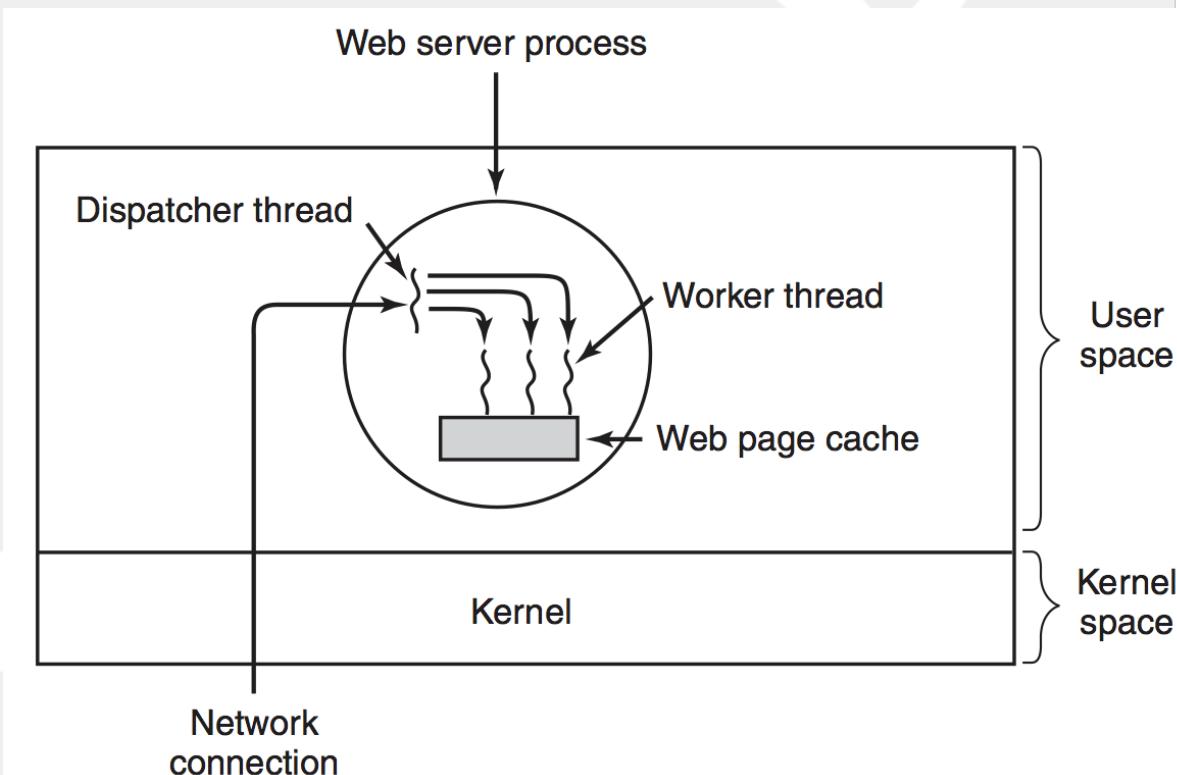
# Motivation for Threads

- Most modern applications are multithreaded
- Why?
  - Applications want to do many different **not so tightly coupled** activities
  - Threads can make such activities happen at the "same" time



# Another Application: Multi-threaded Web Server

- Web servers are multi-threaded to handle many requests in a given amount of time



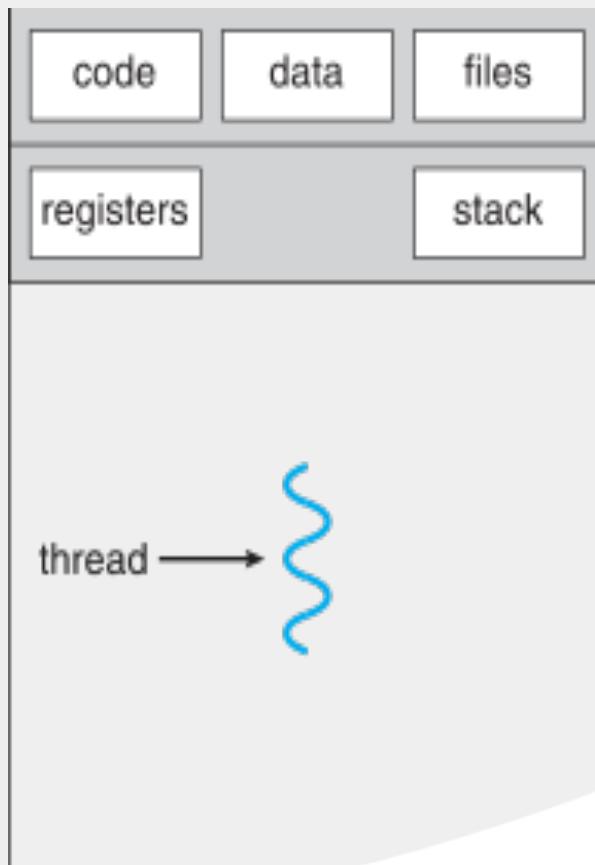
# Why Not Use Processes?

- Processes are heavy-weight
- Threads are light-weight
- Threads are also sharing memory – easier programming compared to multiple processes
- Threads are less fault tolerant
- Threads can introduce lot of synchronization issues (e.g., race conditions) if not done correctly

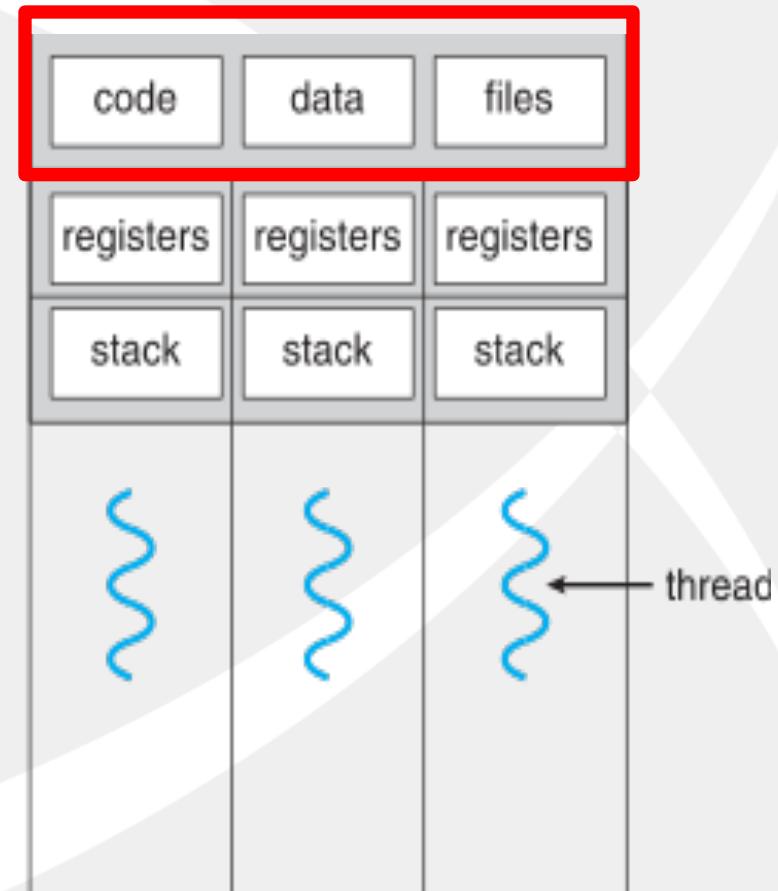
# Why Process Creation Heavy?

- Process creation:
  - ◆ Needs to setup a new address space, allocate resource
  - ◆ Kernel per-process data structures need to be allocated and initialized
- Why threads lightweight?
  - ◆ Threads live within processes
  - ◆ All threads share resources with other threads within the process (minus the stack)

# Single and Multithreaded Processes



single-threaded process



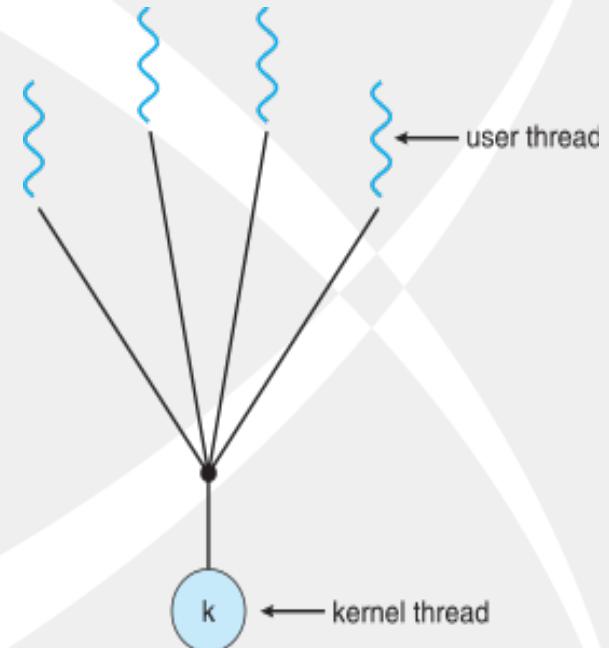
multithreaded process

# User vs. Kernel Threads

- **User threads** - management done by user-level threads library
- Three primary thread libraries:
  - ◆ POSIX **Pthreads**
  - ◆ Windows threads
  - ◆ Java threads
- **Kernel threads** - Supported by the Kernel
- Examples – virtually all general purpose operating systems, including: Windows, Solaris, Linux

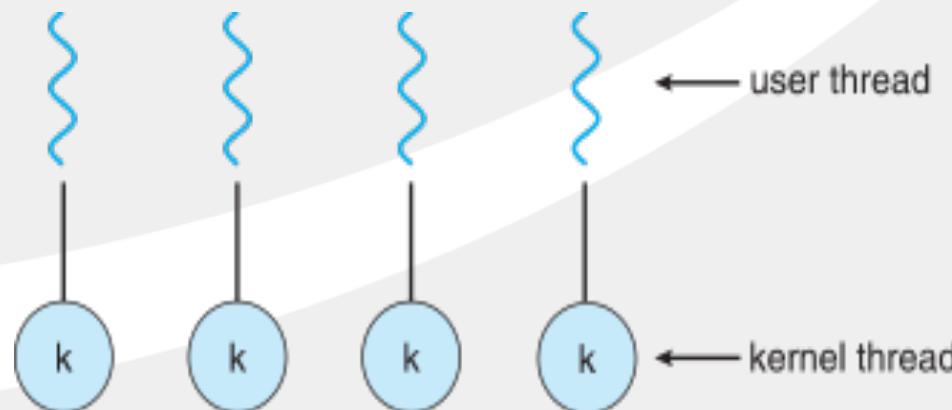
# User Level Threads

- Many user-level threads mapped to single kernel thread
- One thread blocking causes all to block
- Multiple threads may not run in parallel on multicore system because only one may be in kernel at a time
- Few systems currently use this model
- Examples: **Solaris Green Threads**, **GNU Portable Threads**



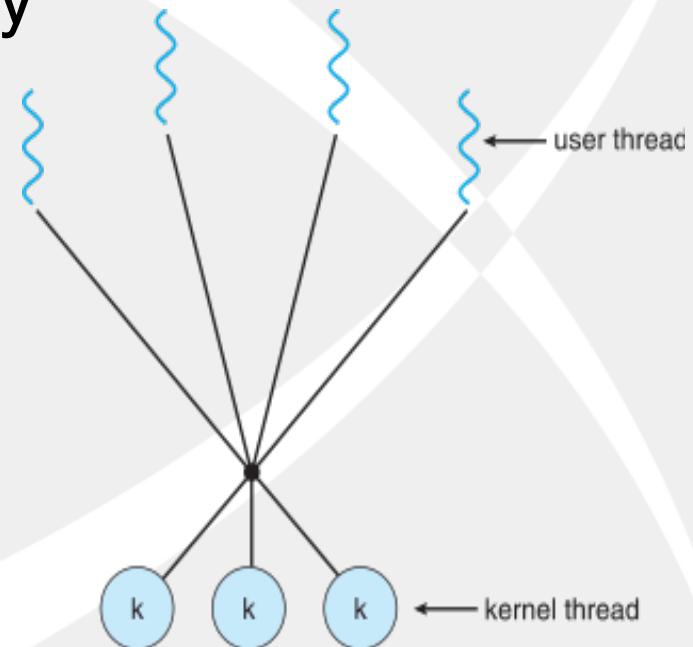
# Kernel-Level Threads

- Each user-level thread maps to kernel thread
- Creating a user-level thread creates a kernel thread
- More concurrency than many-to-one
- Number of threads per process sometimes restricted due to overhead
- Examples: [Windows](#), [Linux](#), [Solaris 9 and later](#)



# Hybrid Threads

- Allows many user level threads to be mapped to many kernel threads
- Allows the operating system to create a sufficient number of kernel threads
- Solaris prior to version 9
- Windows with the *ThreadFiber* package



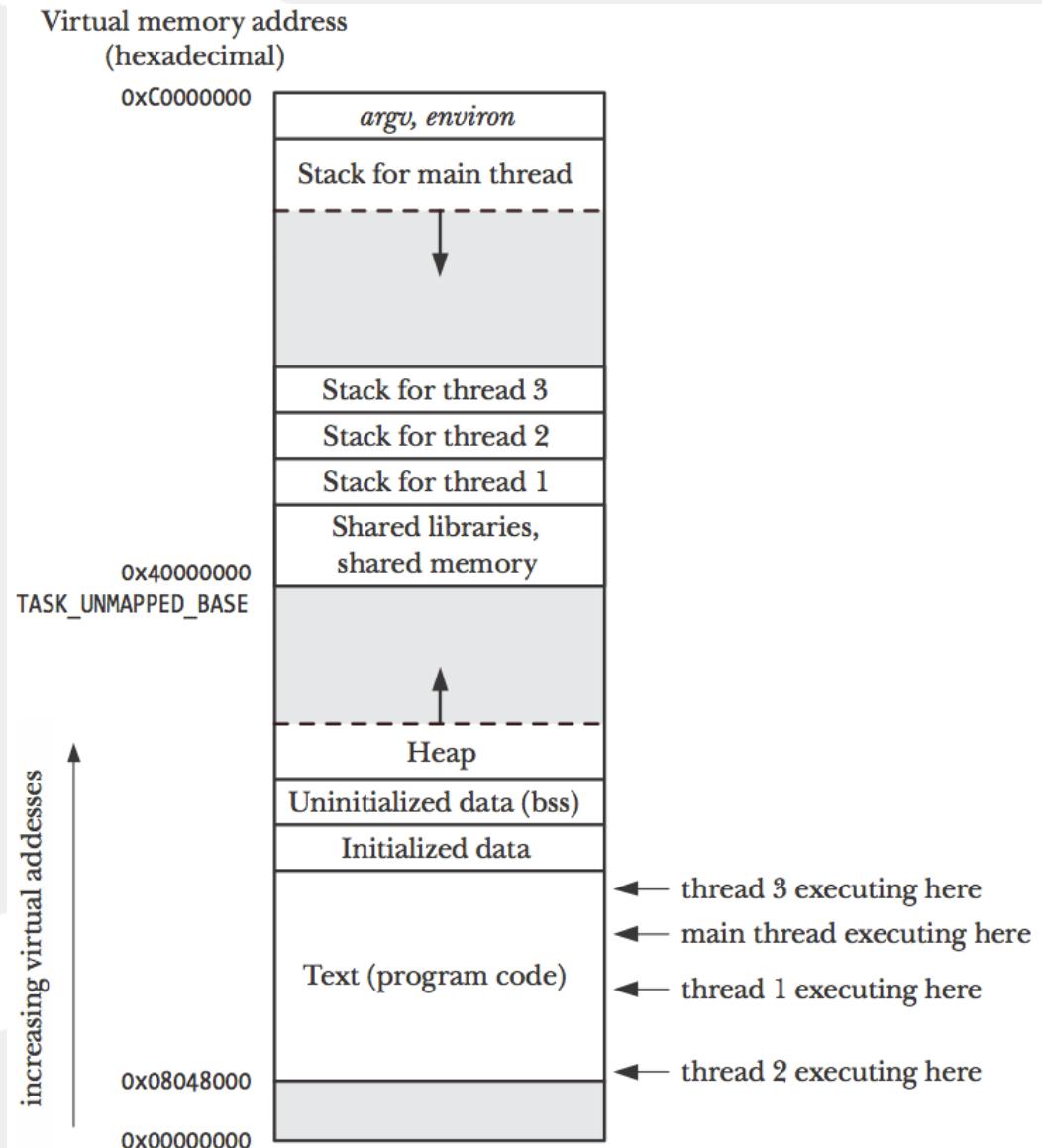
# Thread Libraries

- **Thread library** provides programmer with API for creating and managing threads
- Two primary ways of implementing
  - ◆ Library entirely in user space
  - ◆ Kernel-level library supported by the OS

# Threads in Linux

- Four threads executing in Linux

- ❖ Kernel level threads
- ❖ Threads have specific stacks – thread local storage



# Pthreads

- May be provided either as user-level or kernel-level
- A POSIX standard (IEEE 1003.1c) API for thread creation and synchronization
- ***Specification***, not *implementation*
- API specifies behavior of the thread library, implementation is up to development of the library
- Common in UNIX operating systems (Solaris, Linux, Mac OS X)

# Pthread Creation

- Process has the *main* thread at the beginning

```
#include <pthread.h>

int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                  void *(*start)(void *), void *arg);
```

Returns 0 on success, or a positive error number on error

- New thread continues with start() and main continues with the statement after

# Pthread Termination

- A thread terminates for the following:
  - ◆ The start() function performs a return
  - ◆ Thread calls a pthread\_exit() function
  - ◆ Thread is cancelled using pthread\_cancel()
  - ◆ Any thread calls exit() or main thread returns

```
include <pthread.h>

void pthread_exit(void *retval);
```

# Identities of Threads

- Each thread is uniquely identified by an ID
  - ◆ returned to the caller of `pthread_create()`
  - ◆ thread can obtain own ID using `pthread_self()`

```
include <pthread.h>

pthread_t pthread_self(void);
```

Returns the thread ID of the calling thread

- IDs allow checking if two threads are same

```
include <pthread.h>

int pthread_equal(pthread_t t1, pthread_t t2);
```

Returns nonzero value if  $t1$  and  $t2$  are equal, otherwise 0

# Joining a Terminated Thread

- A thread can wait for another thread using the `pthread_join()` function

```
include <pthread.h>
```

```
int pthread_join(pthread_t thread, void **retval);
```

Returns 0 on success, or a positive error number

- If a created thread is not *detached*, we ***must join*** with it, otherwise “zombie” thread will be created

# Pthread Example

```
#include <pthread.h>
#include "tlpi_hdr.h"

static void *
threadFunc(void *arg)
{
    char *s = (char *) arg;

    printf("%s", s);

    return (void *) strlen(s);
}

int
main(int argc, char *argv[])
{
    pthread_t t1;
    void *res;
    int s;

    s = pthread_create(&t1, NULL, threadFunc, "Hello world\n");
    if (s != 0)
        errExitEN(s, "pthread_create");

    printf("Message from main()\n");
    s = pthread_join(t1, &res);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("Thread returned %ld\n", (long) res);

    exit(EXIT_SUCCESS);
}
```

# Detaching a Thread

- Default – a thread is joinable – another thread is going to retrieve the return state
- If no thread is interested in joining we need to detach the thread

```
#include <pthread.h>  
  
int pthread_detach(pthread_t thread);
```

Returns 0 on success, or a positive error number

thread

# Thread Attributes

- Attributes can be used to set properties of threads – such as detached

```
pthread_t thr;
pthread_attr_t attr;
int s;

s = pthread_attr_init(&attr);                  /* Assigns default values */
if (s != 0)
    errExitEN(s, "pthread_attr_init");

s = pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
if (s != 0)
    errExitEN(s, "pthread_attr_setdetachstate");

s = pthread_create(&thr, &attr, threadFunc, (void *) 1);
if (s != 0)
    errExitEN(s, "pthread_create");
```

# Protecting Shared Variables

- Advantage of threads – can share via global variables
- Must ensure multiple threads are not modifying the variables at the same time
- Use a `pthread_mutex` variable

```
#include <pthread.h>
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);  
int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Both return 0 on success, or a positive error number

# Example Program

```
#include <pthread.h>
#include "tlpi_hdr.h"

static int glob = 0;
static pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

static void *                /* Loop 'arg' times increm
threadFunc(void *arg)
{
    int loops = *((int *) arg);
    int loc, j, s;

    for (j = 0; j < loops; j++) {
        s = pthread_mutex_lock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_lock");

        loc = glob;
        loc++;
        glob = loc;

        s = pthread_mutex_unlock(&mtx);
        if (s != 0)
            errExitEN(s, "pthread_mutex_unlock");
    }

    return NULL;
}
```

```
int
main(int argc, char *argv[])
{
    pthread_t t1, t2;
    int loops, s;

    loops = (argc > 1) ? getInt(argv[1], GN_GT_0, "num-loops");

    s = pthread_create(&t1, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");
    s = pthread_create(&t2, NULL, threadFunc, &loops);
    if (s != 0)
        errExitEN(s, "pthread_create");

    s = pthread_join(t1, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");
    s = pthread_join(t2, NULL);
    if (s != 0)
        errExitEN(s, "pthread_join");

    printf("glob = %d\n", glob);
    exit(EXIT_SUCCESS);
}
```

# Thread Cancellation

- Terminating a thread before it has finished
- Thread to be canceled is **target thread**
- Two general approaches:
  - ◆ **Asynchronous cancellation** terminates the target thread immediately
  - ◆ **Deferred cancellation** allows the target thread to periodically check if it should be cancelled
- Pthread code to create and cancel a thread:

```
pthread_t tid;

/* create the thread */
pthread_create(&tid, 0, worker, NULL);

. . .

/* cancel the thread */
pthread_cancel(tid);
```

# Thread Cancellation...

- Invoking thread cancellation requests cancellation, but actual cancellation depends on thread state

Mode	State	Type
Off	Disabled	-
Deferred	Enabled	Deferred
Asynchronous	Enabled	Asynchronous

- If ~~the thread has cancellation disabled~~, cancellation remains pending until thread enables it

# Thread Cancellation...

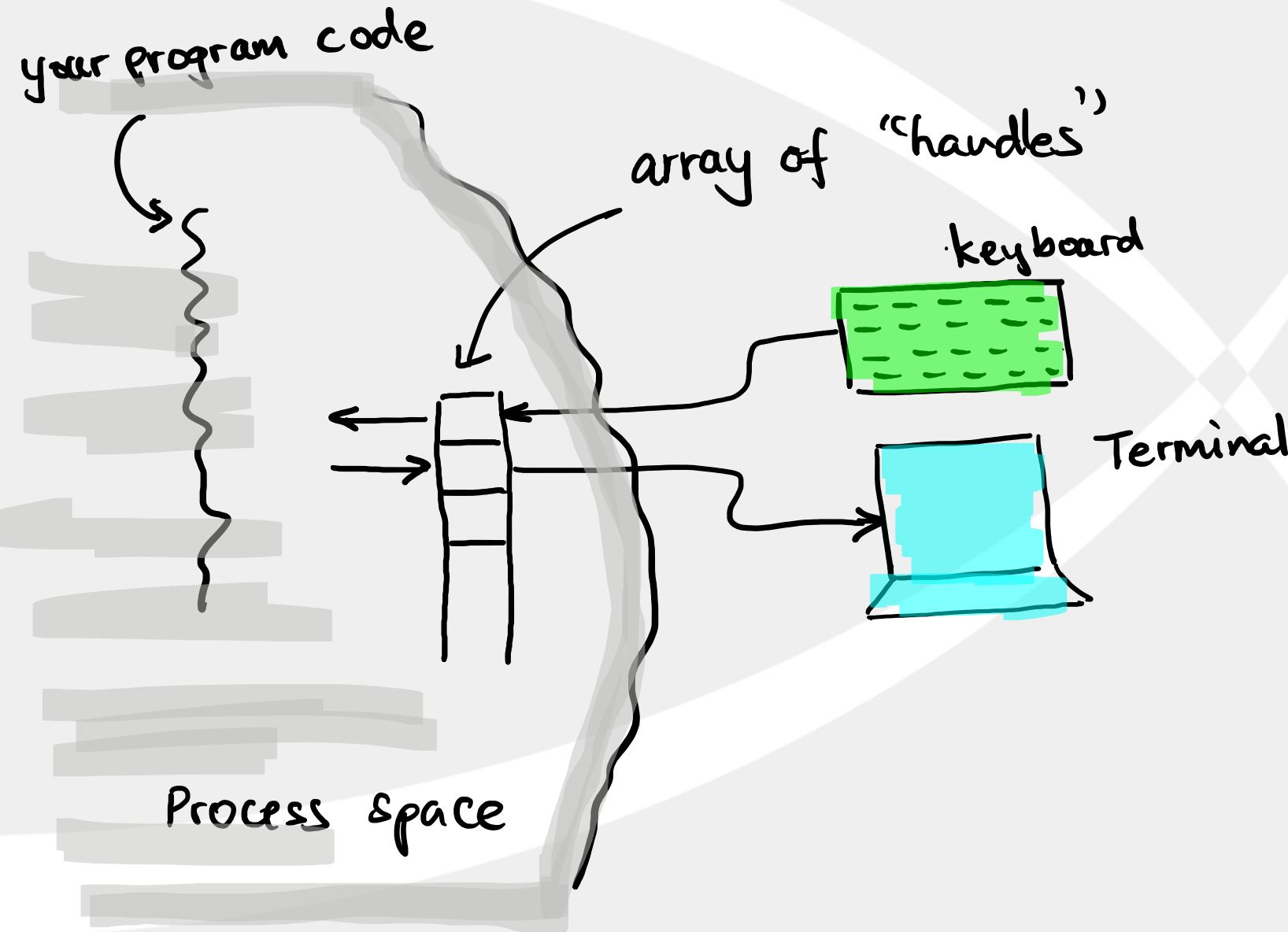
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - ▶ I.e. `pthread_testcancel()`
    - ▶ Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

# Processes and Threads (2)

# Process I/O: How Does it Work?

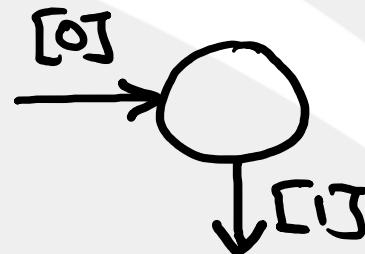
- Each process has an array of “handles”
- Each handle corresponds to an external device
- For example, you want to print something to screen, you write to handle #1
- You want to read something from keyboard, you read from handle #0
- You can setup handles to point to your data file and start reading and writing

# Process I/O: In Pictures



# Process I/O: Simple Example

Read from standard input and write to standard output.



how can we use this program ??

```
main( ) {  
    char buf[BUFSIZE];  
    int n;  
    const char* note = "Write failed\n";  
  
    while ((n = read(0, buf, sizeof(buf))) > 0)  
        if (write(1, buf, n) != n) {  
            (void)write(2, note, strlen(note));  
            exit(EXIT_FAILURE);  
        }  
    return(EXIT_SUCCESS);  
}
```

What does this program do??

# Understand the Simple Example

Read from stdin [0] until no input

(end of input)

```
main( ) {  
    char buf[BUFSIZE];  
    int n;  
    const char* note = "Write failed\n";  
  
    while ( (n = read(0, buf, sizeof(buf))) ) > 0 )  
        if (write(1, buf, n) != n) {  
            (void)write(2, note, strlen(note));  
            exit(EXIT_FAILURE);  
        }  
    return(EXIT_SUCCESS);  
}
```

write to stdout [1] and check  
for error.

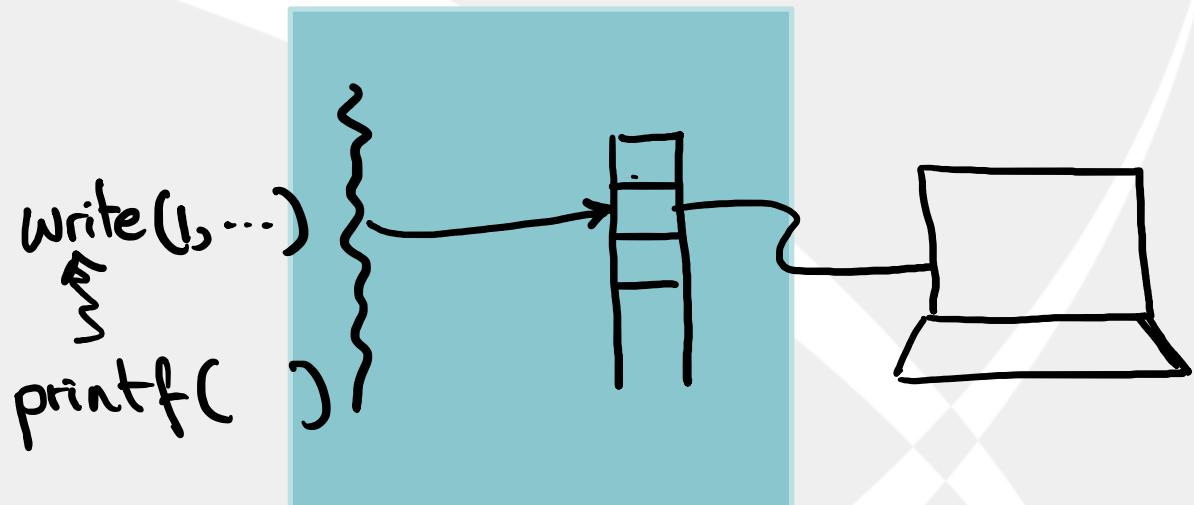
# Process I/O: “Handles”

- We refer to the “handles” as **file descriptors**
- File descriptors
  - ◆ A array structure maintained by the kernel for each process
  - ◆ Held in the kernel memory and process can modify them through syscall

# Example

Consider the following:

```
char *p =  
"hello\n";  
  
write(1, p, 6);  
  
printf("%s\n",  
"hello");
```



`printf` is a high level  
library routine built  
on top a low level syscall  
like `write()`

# Description of the Example

```
write(1, str, strlen(str));
```

- This asks the kernel to write the string (str) to the device pointed to by file descriptor #1.
- By default file descriptor #1 is pointing to the standard output (screen)

# At Process Creation

- File descriptor #0 (stdin): is the keyboard or whatever the standard input is
- File descriptor #1 (stdout): is the screen, printer or whatever the standard output is
- File descriptor #2 (stderr): is the error device which is mostly the same as stdout
- By having error separate from stdout, we can separate them out as needed

# File Manipulation

```
int fd = open("filename", ...)
```

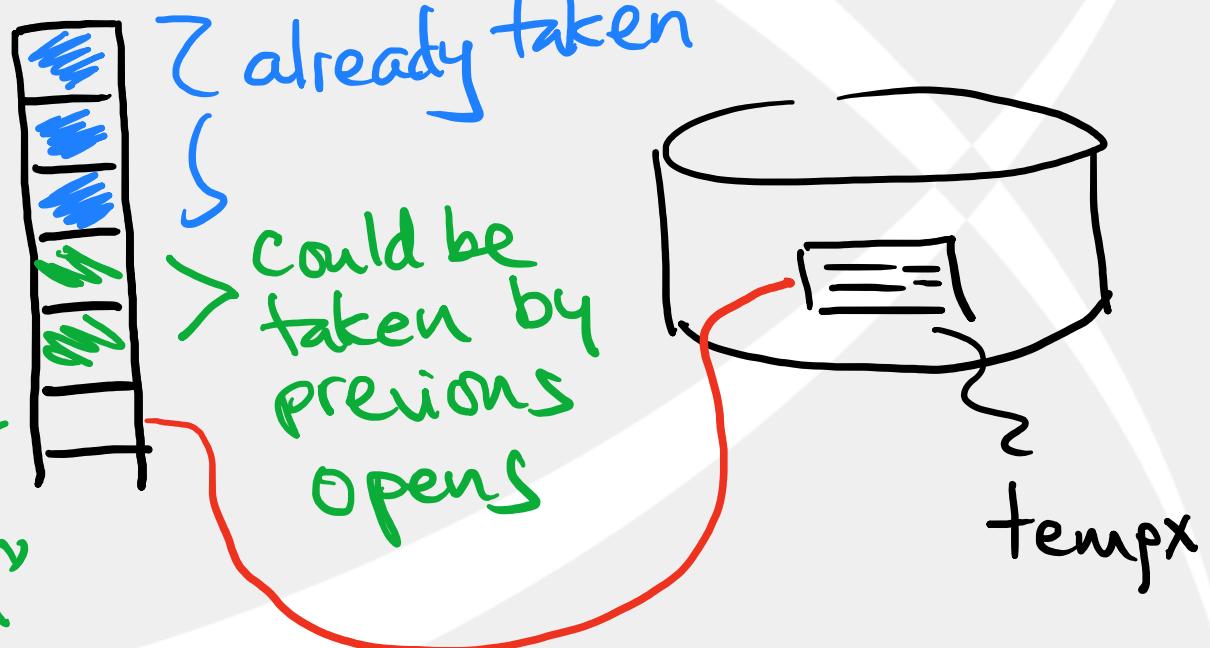
- The value returned by the `open()` is file descriptor pointing to the given file
- You can think that the OS did a “open” for you on `stdin`, `stdout`, `stderr` when it created the process

# File Manipulation: In Picture

`fd = open ("tempx", ...)`  
first available `fd`.

$\downarrow$   
`fd = 5`

returned  
for this "open"



read/write to `fd #5`  
results in read/write to `tempx`

# Output Redirection: Homework

- Write a simple C program that does the following
  - ◆ It has few printf() statements that are printing some messages to the screen
- Compile and run the program – nothing exciting
- Now “redirect” the output by redirecting the standard output to a file – temp.txt
- Tip: close(1) and open(“temp.txt”..) afterwards; why would this do the trick?

# Little Digression

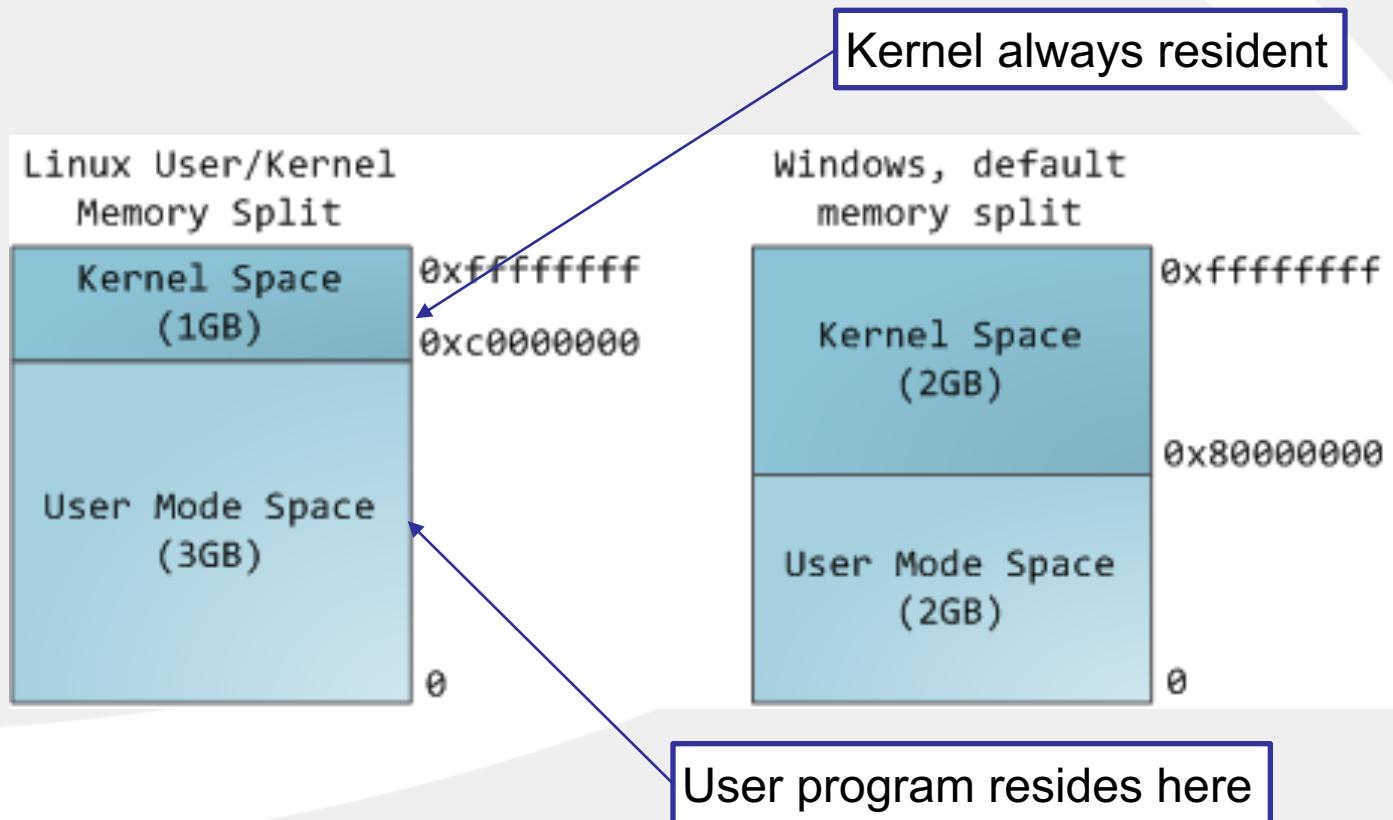
- We already discussed that a process has many elements
- One of the questions we encountered is where all the elements go
- Some of them go inside the processor
- Others go in the memory
- Obviously the memory needs to be organized in some way to put them

# Memory from a Process' Viewpoint

- From a process' viewpoint, memory is a large array
- There are many processes in an OS
- Each process has a *logical view*
  - ◆ looks like it owns half or a big chunk of the memory
  - ◆ other half is held by the kernel
- Where are the other processes? We will talk about that later!

# Address Spaces

- Instead of sharing the memory space – give each process the full address space

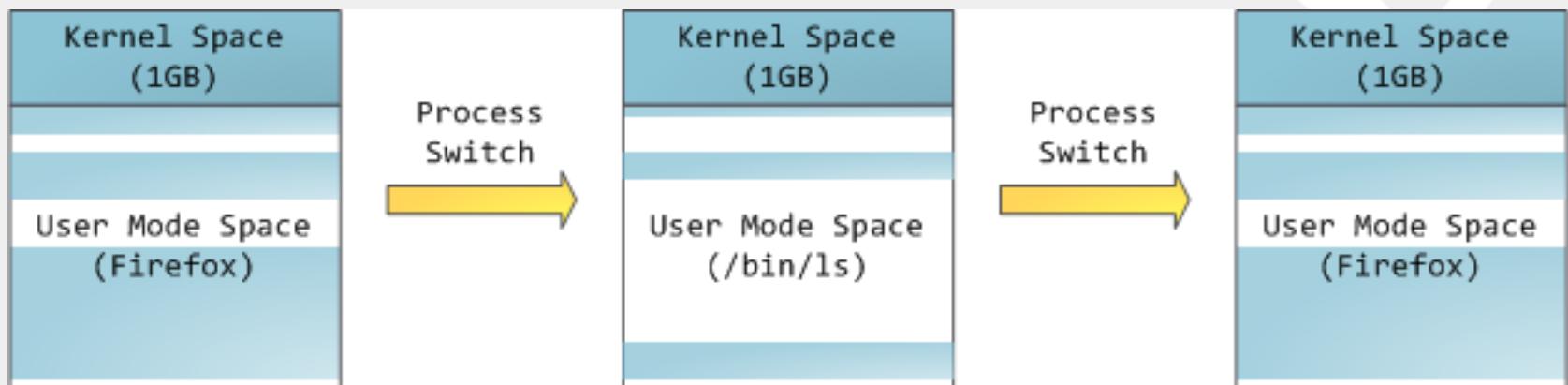


# Address Space

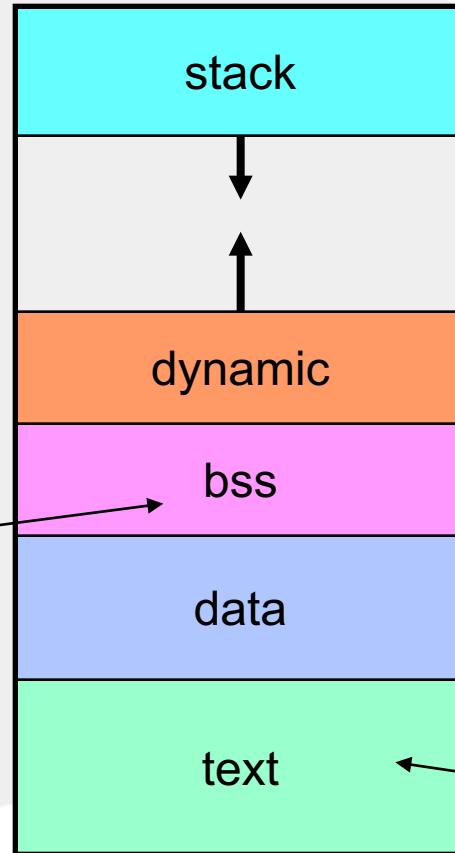
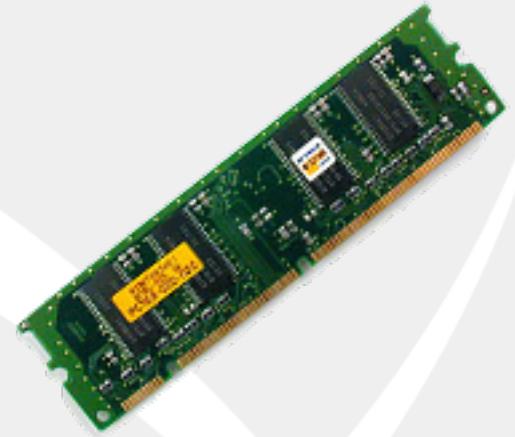
- When a process runs, all the addresses it could generate (full address space) belong to it – not shared with any other process
- Catch – **part of it is taken by the kernel** – the space mapped to the kernel – is persistent
- Using the kernel space – a process could communicate with other processes, how?

# Address Space Switching

- Address space switching happens with a process switch



# Address Space: Details of the User Space

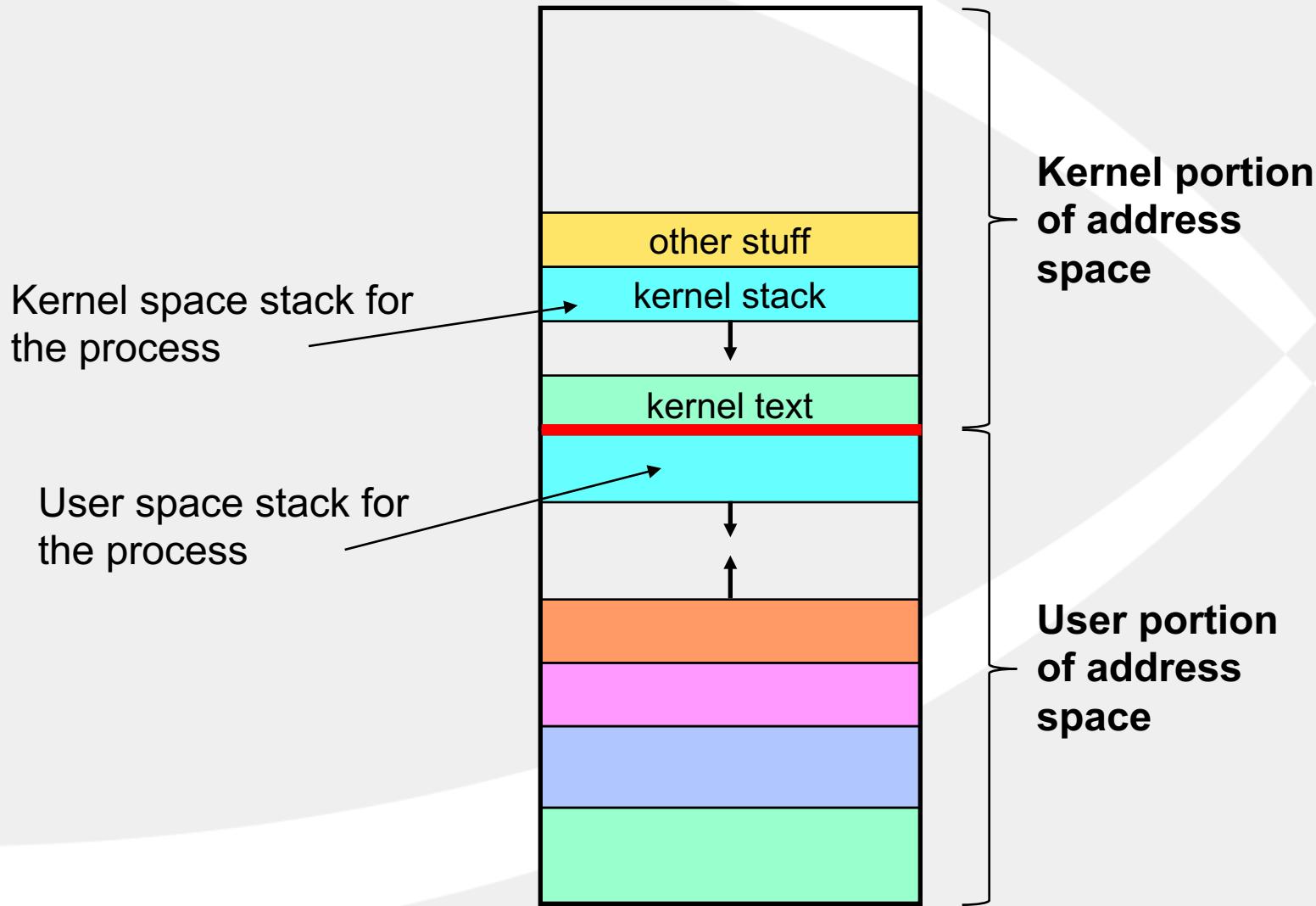


Block Started by Symbol:  
Includes space for uninitialized  
variables

**Read only** segment  
that contains the  
program instructions

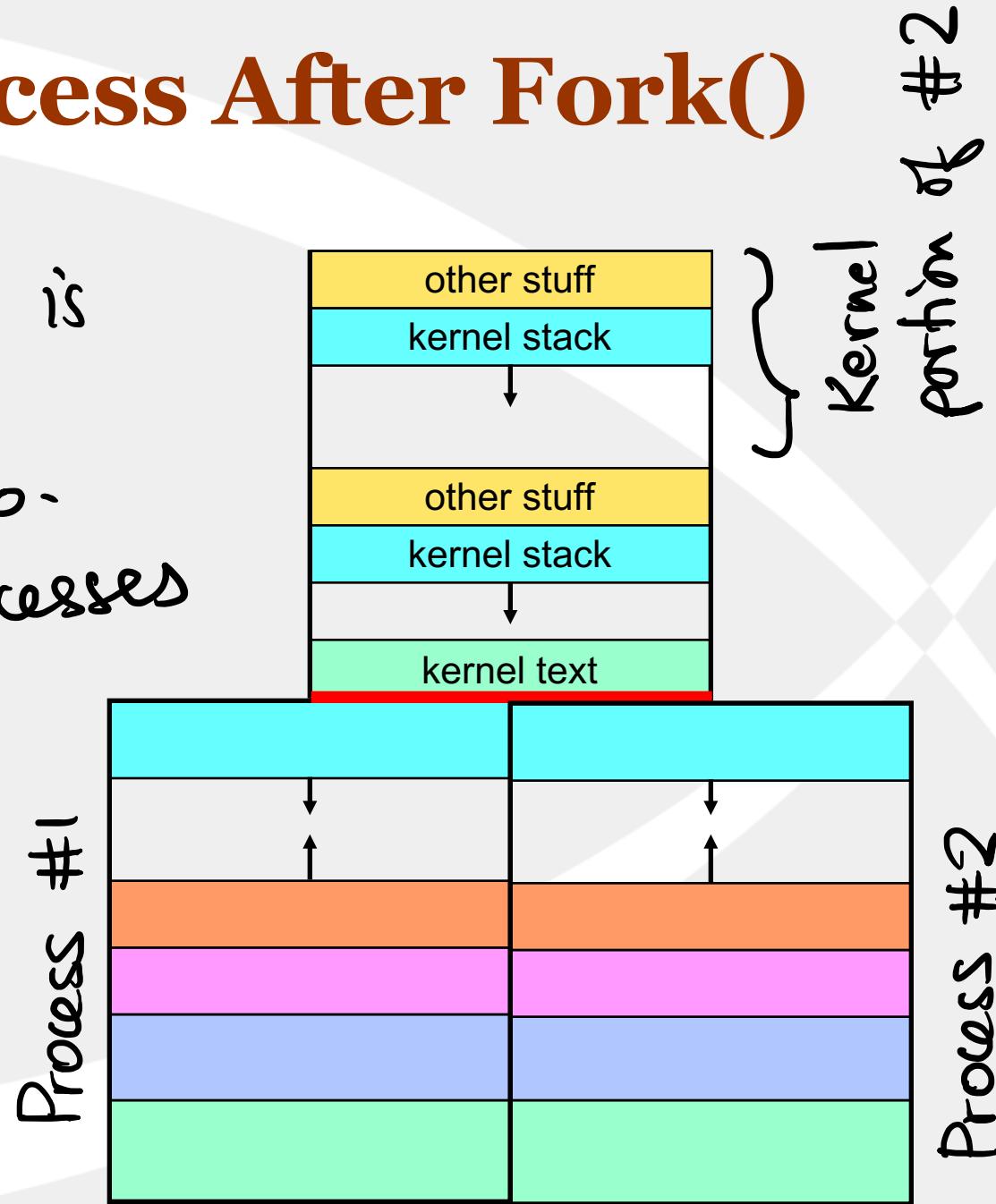
Address space occupied by user programs when  
**address space is NOT randomized**

# Process Address Space



# Process After Fork()

Kernel Space is  
suitable for  
sharing info-  
processes



# Something Interesting In Shells

- Lets consider interesting shell functions
- Output redirection
  - ◆ We run a program, the program writes something to the screen
  - ◆ We want to redirect the output to a file so that we can print it or look at it carefully
  - ◆ How do we implement this functionality in a shell?

# Input Redirection

- Like the output redirection but dealing with the input
- We have a program that reads from the terminal (e.g., keyboard)
- We want to run the program in batch mode – so no keyboarding
- We want to supply the input using values from a file
- How to implement this functionality?

# More Interesting

- We want to make one command's output reach another command's input
- How do we implement the command piping?
- That is send the output of one process as input of another process?
- Hint: assume first process is sending output to stdout, second process is reading the stdin

# Implement Output Redirection

## ■ Key observations:

- ◆ A newly created process by default has has stdin, stdout, and stderr wired as described
- ◆ Output will go through stdout by default
- ◆ Change the stdout wiring
- ◆ Load the program into the process and execute it

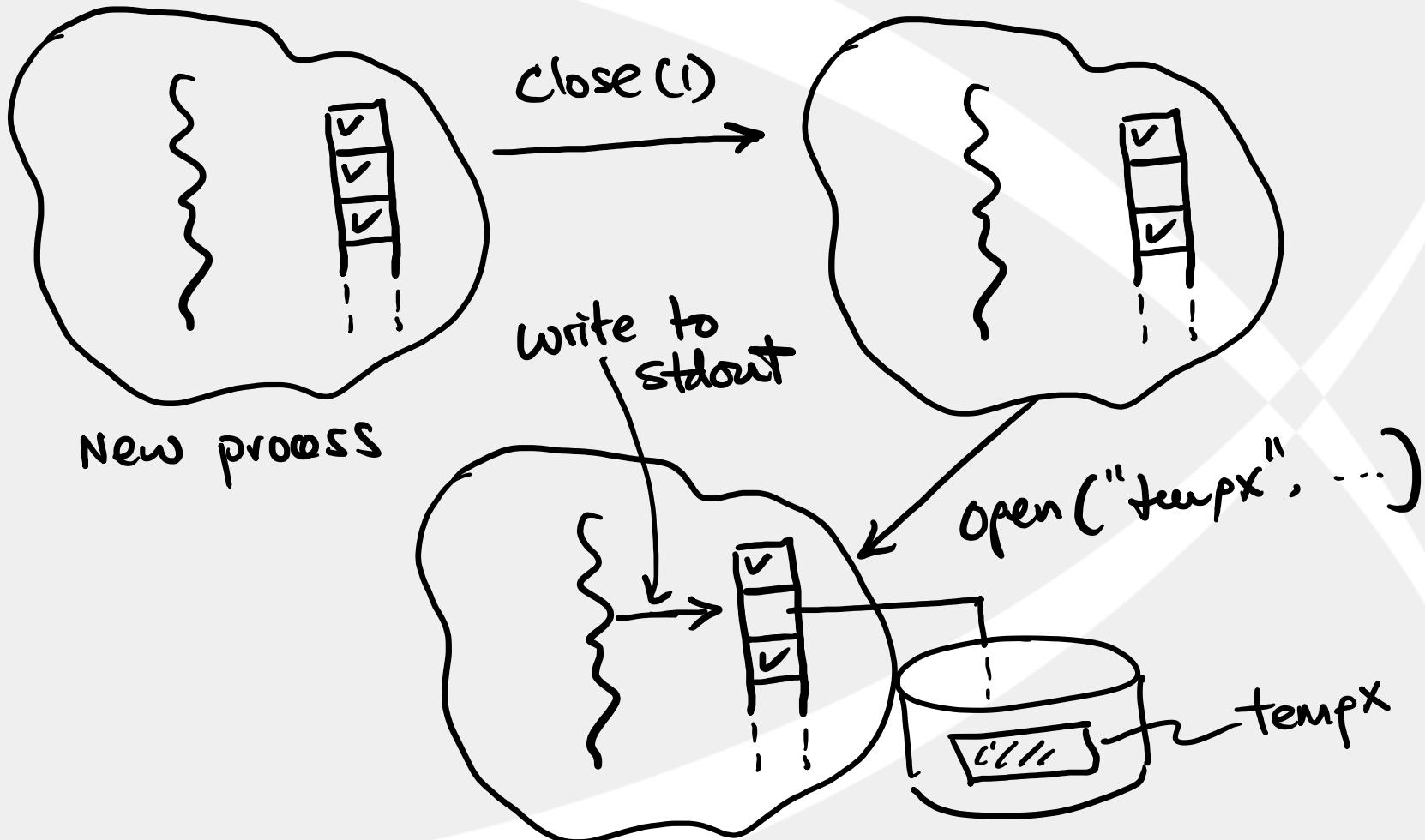
# Important Note on Application Launching

- When you launch an application through GUI or terminal
  - ◆ New process is created by forking a previously running process (e.g., shell)
  - ◆ Application is loaded into the new process and execution resumes at the “start” point

# Implement Output Redirect

- Goal:
  - ◆ Stdout should point to the redirected file
- How to achieve?
  - ◆ Hint: open() always gives the first available file descriptor
  - ◆ close(1), so file descriptor is available for sure
  - ◆ open() now returns file descriptor #1

# Redirection In Pictures



# Input Redirection: Homework

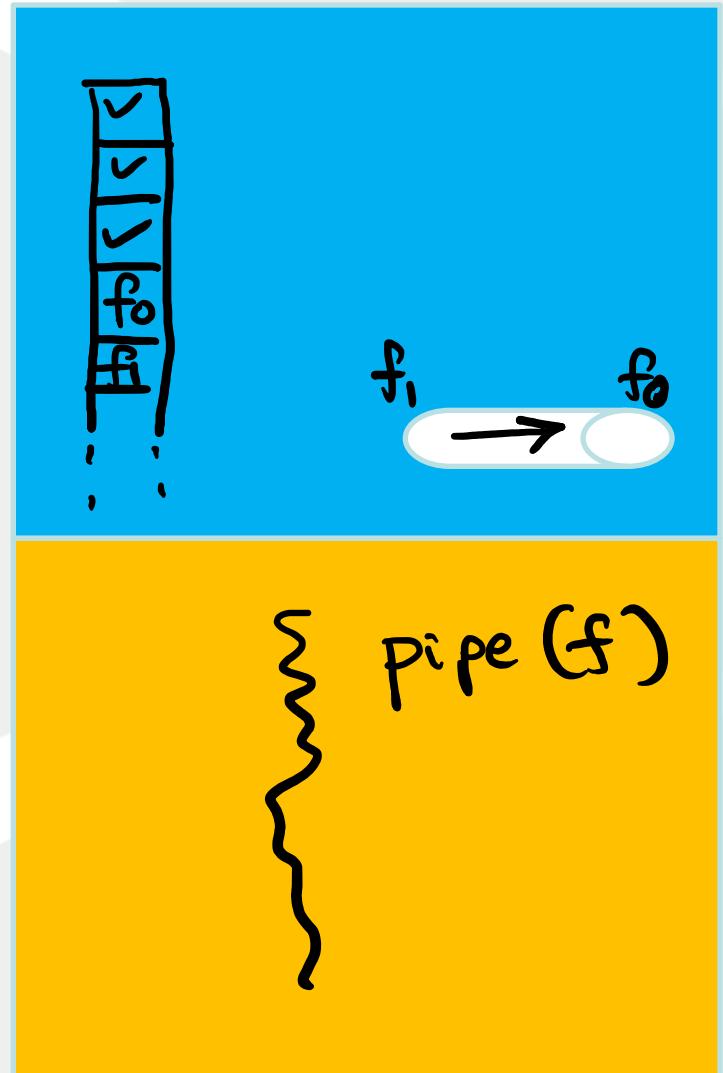
- Implement input redirection

# Command Piping

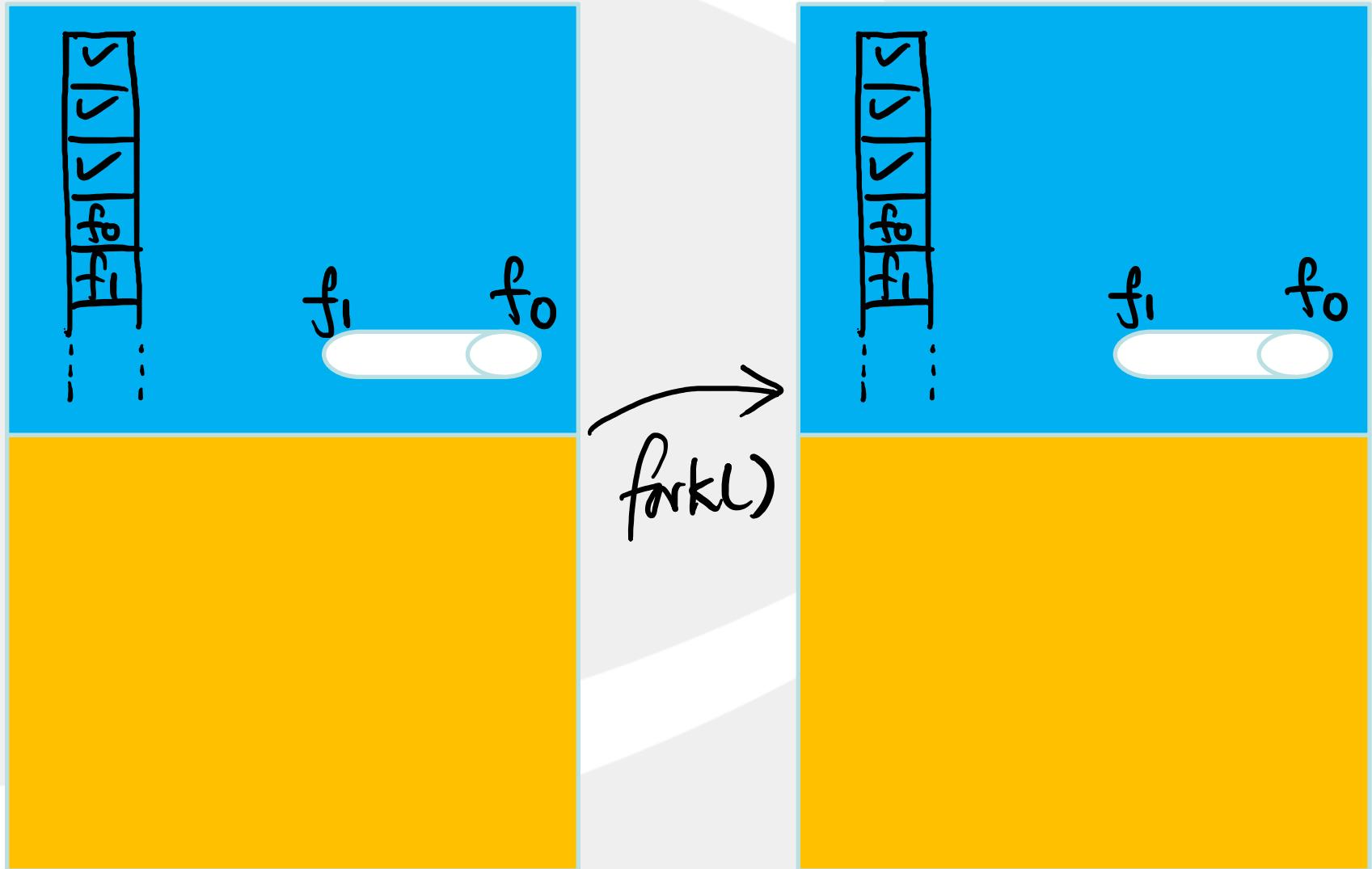
- Goal:
  - ◆ Connect the output of a process to the input of another process
- Problem:
  - ◆ Processes are independent of each other
- Idea:
  - ◆ Do the transfer through the kernel space

# Step 1: Create a Pipe

- Use the pipe() system call and create a pipe
  - ◆ Has two end points
  - ◆ Allows unidirectional data flow
  - ◆ Write end and read end



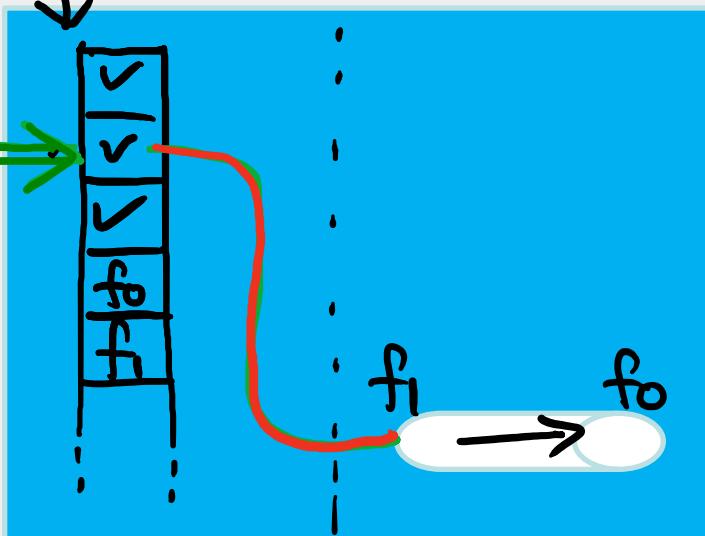
# Step 2: Fork the Process



# Step 3: Rewire the File

## Descriptors

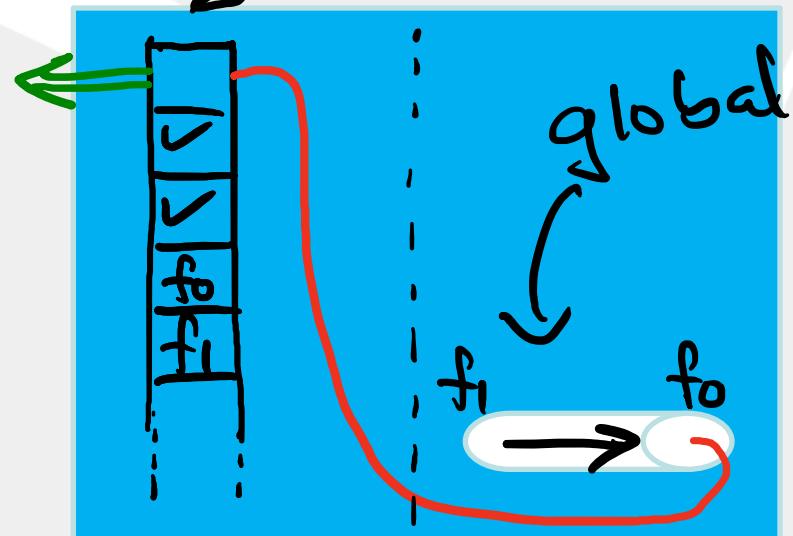
per process



close ()  
dup ( $f[0]$ )

Process #0

per process



close (0)  
dup ( $f[0]$ )

Process #1