OPERATING SYSTEMS (COMP 310/ECSE 427) – Alternate Midterm – March 12, 2013

Name: **Student ID:**

Provide **brief answers to all** questions. This exam is worth **100 points** in total.

1. This question has multiple parts **[20 points]**
    a. What are the major issues faced when developing a scheduler for symmetric multiprocessors? Briefly explain how the O(1) scheduler addresses these issues.

There are two major issues: contention for the task queue and cache footprint management. The O(1) scheduler addresses these issues by having a local task queue per processor. This allows concurrent access and also the tasks remain local to the processors as much as possible.

    b. Give a memory configuration and sequence of memory requests for which the best-fit would outperform first-fit.

Consider a memory configuration with holes in decreasing sizes. Suppose both algorithms are scanning from address 0 and we have the following holes (starting address, size): (1000, 500), (2000, 300), (3000, 100). We get the following requests for memory allocation: 300 and 400 (in that order). Best fit would put the first request in the hole starting at 2000 and the second one at the address starting at 1000. First fit would put the first request at the hole starting at 1000. At this point the largest hole would be of size 300 – the one starting at 2000. The second request cannot be allocated.

    c. Buddy algorithm is used to manage dynamic memory allocation. The memory region allocated by the Buddy algorithm has a maximum capacity of 64 Kbytes. Suppose the memory region starts at 0x800000. Find all possible starting addresses for the block allocated by Buddy for a request of size 5 Kbytes. What is the internal fragmentation of an allocation performed by Buddy?

Starting addresses are 0x800000, 0x802000, 0x804000, 0x806000, 0x808000, 0x80A000, 0x80C000, 0x80E000. The internal fragmentation is 3K for an allocation performed by the buddy.

    d. What is the maximum internal fragmentation you can get with the Buddy in the above situation?

The largest block that could be allocated by the buddy is 64Kbytes. So, the maximum internal fragmentation is 32767 bytes.

2. Deadlock problems [**25 points**]
    a. State the necessary and sufficient conditions for a deadlock to happen.

Circular wait.

Hold and wait.

No preemption.

Mutual exclusion.

    b. A system has four processes and five allocatable resources. The current allocation (or the "Hold" matrix) is given by [1 0 2 1 1; 2 0 1 1 0; 1 1 0 1 0; 1 1 1 1 0]. The claim matrix (or "Max" matrix) is given by [1 1 2 1 3; 2 2 2 1 0; 2 1 3 1 0; 1 1 2 2 1]. The available vector is given by [0 0 x 1 1]. What is the smallest value of x for which this is a safe state?

The available vector was corrected to [0 0 x 1 2] in the exam hall. The smallest x is 1.

    c. A computer has six tape drives, with *n* processes competing for them. Each process may need two drives. For which values of n is the system deadlock free?

The only way to have a deadlock is to have 6 processes and each one hold a tape. Suppose we have just 5 processes. At least one process should get 2 tapes. So it can complete and then two more processes can get 2 tapes each. So they can complete too. Therefore, the system will be deadlock free for n < 6.

    d. Briefly discuss at least two ways of implementing deadlock prevention.

First way is by ordering the resources in an arbitrary order. When multiple resources need to be acquired, we should acquire them in the predefined order. Because all processes are respecting the same order, circular wait is prevented. So we have prevented deadlocks.

Second way is by allocating all resources in one single allocation. If a resource in the group is not available, then none of the resources are allocated. This invalidates the hold-and-wait condition. So the deadlock is prevented.

3. Synchronization problems [**30 points**]
    a. Implement a barrier synchronization mechanism using semaphores. Your barrier should be reusable (can be called multiple times). You need to use minimum number of semaphores.

```
Semaphore bwait = 0
Semaphore mutex = 1
int pcount = N

wait(mutex)
        N = N -1
```

```
        if (N > 0) {
                signal(mutex)
                wait(bwait)
        } else
                signal(bwait)
        N = N + 1
signal(mutex)
```

b. Implement a monitor using semaphores. In this monitor, signal() wakes up all the threads that are waiting on the condition variable on which the signal was invoked. Also, the signaling thread does not wait. Instead, the thread(s) woken up by the signal wait until the signaling thread leaves the monitor. Although signal() wakes up all threads, they are prevented from being active within the monitor at the same time – which would violate the condition of the monitor.

```
Semaphore  mutex = 1
Semaphore next = 0
Semaphore xsem = 0
int xcount
int nextcount

wait(mutex)
<body of a function>
if (nextcount > 0)
        signal(next)
else
        signal(mutex)

cwait(x)
        xcount++
        if (nextcount > 0)
                signal(next)
        else
                signal(mutex)
        wait(xsem)
        xcount—
        nextcount++
        wait(next)
        nextcount—
```

```
csignal(x)
        for I from 0 to xcount -1
                signal(xsem)
```

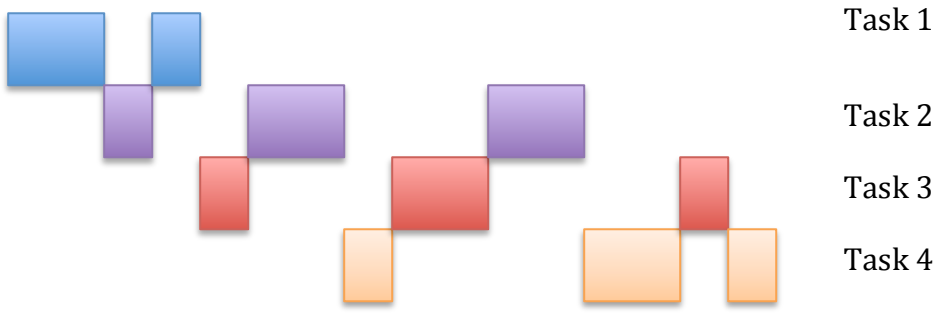
4. Scheduling problems [**25 points**]
    a. Provide a sequence of 10 tasks (specify their service times and arrival times) that would give the same average completion times for shortest job first and first-come first served. If you cannot find such a task sequence explain why it is not possible.

Any task sequence where tasks come in **non decreasing** sizes would give the same average completion times for both SJF and FCFS. For example, (0, 10), (5, 10), (10, 15), (15, 20), (15, 20), (20, 30), (20, 30), (25, 35), (30, 40), (50, 60) would give the same average completion times for both.

    b. Rate monotonic scheduler is asked to schedule a task with deadline 50 and service time 30. Give a second task it cannot schedule but an earliest deadline scheduler can schedule.

Consider another task with a deadline of 75 and service time of 25. This task has lower priority than the first task. The RMA will schedule the first task every 50 time units. In the first 75 units, there will be only 20 free time units. Therefore, RMA cannot schedule the second task. The earliest deadline first scheduler can schedule both tasks. In the first 150 time units, we have 30 + 25 + 30 + 25 + 30 = 140 time units. Therefore, EDF can find a feasible schedule.

    c. Schedule the following tasks using a multi-level feedback queue scheduler. There are four levels in the scheduler and the maximum quantum increases from 1 (at the top) to 8 at the bottom. Task 1 (arrival 0, service 3), Task 2 (arrival 2, service 5), Task 3 (arrival 4, service 4), Task 4 (arrival 6, service 4)



OPERATING SYSTEMS (COMP 310 / ECSE 427) – Midterm – February 27, 2014

Name: Solutions **Student ID:**

Part I –Some multiple-choice questions can have **more than one correct** answer. Circle *all and only correct* answers. (Answer **all 10 questions – 6 points each**)

1. What are two important differences between a system call and a function call?

1. System call runs code part of OS

2. System call runs in privileged mode

2. How does address space virtualization (or memory virtualization) help multi-processing?
   a. Allows the different processes to load their segments to the same virtual address locations
   b. Isolates the address space between two processes that are resident in the memory at the same time
   c. Allows the CPU to access the memory faster
   d. Resolves memory contention in a fair manner
   e. None of the above

3. Consider a process that is being context switched out of the CPU (i.e., the process is getting evicted from the CPU and will be scheduled in a later point in time). Which of the following should be always true for the process.
   a. The process will never run on the CPU again
   b. The process is waiting on a semaphore
   c. The process is waiting for a disk operation to complete
   d. The process has been using too much CPU time
   e. None of the above.

A little confusing...

} One possible answer

Another choice

4. Two processes are created by an application. Which of the segments associated with the processes could be same between the two processes?

The text segment could be the same.

5. A process has multiple threads. The threads are implemented at the kernel level. Which of the following statements are true?
   a. The text segment is common across all the threads
   b. The data segment is common across all the threads
   c. All threads share the same stack for all their activities such as function invocation.
   d. The memory map segment is different for the different threads

goes on per-thread Stack

6. A process and its child process are writing their outputs to a file. The output from the parent and child processes do not clobber each other – that is, do not overwrite each other. Which of the following is true regarding the implementation of this feature?
   a. The parent and child processes have the same file descriptor table
   b. The parent and child processes are synchronized so that one writes after the other
   c. The file system table is common to both processes
   d. The parent and child processes have their own entries in the file system table corresponding to the open file each have for outputting data
   e. None of the above
7. A program takes 200 seconds to run. An enhancement is applied to a portion of the program to make it run faster. Only 30% of the program as measured in terms of run time cannot be enhanced by this modification. The modification can improve the run time by a factor of 10. What is the overall speedup?

Enhancement applied to 140s; New runtime = $\frac{140}{10} + 60$

Speedup = $\frac{200}{74} = 2.7$

8. Consider two ways of implementing threads: user-level versus kernel-level. Which of the following statements are true?
   a. With user-level threads blocking system calls can stop the whole application – that is no part of the application is running
   b. With kernel-level threads system calls take longer to execute
   c. With user-level threads the execution time of the application can be shorter because the context switching overhead is small
   d. With kernel-level threads more CPU time is actually used by the application and it could run faster
   e. None of the above
9. Provide the Peterson's algorithm for synchronizing two processes.

Process 0

```
turn = 1
flag[0] = true
while (turn == 1 &&
       flag[1]);
<.....>
flag[0] = false
```

Process 1

```
turn = 0
flag[1] = true
while (turn == 0 &&
       flag[0]);
<.....>
flag[1] = false
```

10. What is priority inversion? Briefly explain.

A low priority process is able to prevent a high priority from executing by holding a lock that is needed by the high priority process. Low priority process is unable to release the lock because the CPU is held by the high priority process busy waiting.

Part II – Long form Questions (Answer **both questions – provide the shortest possible answer**)

1. **(20 points)** Consider the readers and writers problem. We have many readers and writers and readers have priority. A writer needs to wait until all readers have completed before it could start. A reader need not wait for all writers. All readers can access the data object concurrently. Writers have to access the object serially – one after the other. Provide a monitor that provides the following update functions.

```
reader () {
    while(true) {
        // Do something
        get_reader_lock()
        // Do reading
        release_reader_lock()
        // Do something
    }
}
```

```
writer () {
    while(true) {
        // Do something
        get_writer_lock()
        // Do writing
        release_writer_lock()
        // Do something
    }
}
```

```
Monitor RWlock {
condvar RWQueue
bool reading = false
bool writing = false
int readerCnt = 0

get_reader_lock() {
    readerCnt ++
    if (reading)
        return;
    else if (writing)
        RWQueue.wait();
    reading = true
    return;
}

release_reader_lock() {
    readerCnt--
    if (readerCnt == 0)
        reading = false
        RWQueue.signal()
}
```

```
get_writer_lock() {
    if (reading ||
        writing)
    RWQueue.wait();
    writing = true;
}

release_writer_lock() {
    writing = false
    RWQueue.signal();
}
```

changed to [4 3] in the exam

Give full credit students who did with old value...

2. **(20 points)** We have four processes and two different types of resources in a computing system. The maximum claim matrix (C) and current allocation (H) are given below. The availability vector is [3, 2]. Using Banker's algorithm find:
   a. A value for x and a value for y for which the system is safe.
   b. A value for x and a value for y for which the system is unsafe.

Note: there is no unique answer.

$$C = \begin{bmatrix} 8 & 9 \\ x & 6 \\ 7 & 5 \\ 6 & 4 \end{bmatrix}, \quad H = \begin{bmatrix} 3 & y \\ 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix}$$

$$\text{Need} = \begin{bmatrix} 8 & 9 \\ x & 6 \\ 7 & 5 \\ 6 & 4 \end{bmatrix} - \begin{bmatrix} 3 & y \\ 1 & 2 \\ 2 & 2 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 9-y \\ x-1 & 4 \\ 5 & 3 \\ 4 & 3 \end{bmatrix} \begin{matrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{matrix}$$

Avail [4 3]

P_3 can be completed.

We have [6 4]

P_2 can be completed

We have [8 6]

There are other possible answers as well

$x = 15$ (safe no matter value of y)

$x = 9$ (safe no matter the value of y)

Unsafe $x = 15$, $y = 5$

Safe $x = 9$ $y = 7$

OPERATING SYSTEMS (COMP 310/ECSE 427) – Midterm – February 28, 2013

Name: **Answers** **Student ID:**

Provide **brief answers to all** questions. This exam is worth **100 points** in total.

1. This question has multiple parts **[20 points]**
    a. Briefly explain the necessity for splitting the address space and maintaining a kernel part and a user part in a modern operating system.

Splitting the address space into kernel and user part allows the kernel part to reside in the memory while the user part keeps changing with each context switch. Also, the split allows us to have different protection levels for the kernel part. To access the kernel memory, the process got to run at a higher privilege level, which allows protection of the kernel address space.

    b. Suppose a computer is running two applications A and B. Show the view of the virtual address space at different times while running the two applications.

[Kernel Address Space | Process A for User Space] while running process A
[Kernel Address Space | Process B for User Space] while running process B

    c. Why do we need two stacks per process?

One stack is used while the process is in user mode. The other stack is used while the process is in kernel mode. We need a separate kernel mode stack (that is held in the kernel address space) so that a process running in user mode cannot corrupt it.

    d. What is busy waiting? How can busy waiting lead to priority inversion?

Busy waiting is when a process that is unable to take a lock is repeatedly checking whether it can take the lock. Because processes do not go into waiting state when they are unable to take lock, we can have the following scenario. Suppose process X is busy waiting to get into a critical section, while process Y is occupying the critical section and X has higher priority than process Y. With a priority based scheduling policy, process X will not relinquish the processor for process Y to get out of the critical section. So, process X (high priority) is prevented from making progress by process Y (lower priority).

    e. What is spin lock? Why would an operating system use spin lock inside the kernel instead of blocking?

Memory variables are used as locks inside the kernel and hardware provided instructions such as Test-And-Lock or Compare-And-Swap is used to busy wait on the correct value on the memory variable. This is called spin lock. Operating systems use spin locking instead of blocking, it is more efficient to spin lock. When locks can be obtained in short time, it is better to spin lock, which is the case inside the kernel.

2. Deadlock problems [**25 points**]
    a. State the necessary and sufficient conditions for a deadlock to happen.

Circular wait.

Hold and wait.

No preemption.

Mutual exclusion.

    b. You are allocating resources for four processes using Banker's algorithm. There are eight units of resource of a single type in the system. The available at any time is less than eight because some might be already allocated. At a given time the allocation vector (hold vector) looks like [1, 0, 5, y], where y is unknown. The "Max" or "Claim" vector is [3, 2, 7, x], where x is another unknown. Find the minimum x (0 is a valid number) for which the system could go into unsafe state. Show the allocation vector at that point. Assume resources are allocated one unit at a time.

You could have the system going into unsafe state for x = 0. The allocation vector at that point looks like [2, 1, 5].

    c. Consider a situation where N processes are sharing M resources that can be reserved or released one at a time. The maximum claim (the **Max** vector in the slides) that is made by any process does not exceed M, and the sum of all maximum claims is less than M + N. Show that deadlock cannot occur. **Hint:** Use the banker's algorithm. At deadlock all resources are allocated because resources are allocated one at a time.

Total claim = Total Needed + Total Allocation (from banker's algorithm).

It is given that total claim < M + N.

Also, total allocation = M.

Therefore, Total Needed < N.

That is, at least one process does not need anything more. So, deadlock cannot occur.

3. Synchronization problems [**30 points**]
    a. Implement a barrier synchronization mechanism using a monitor. In barrier synchronization, a process calling the barrier() waits until all the processes have called the barrier(). Your barrier should be reusable (can be called multiple times).

```
Monitor Barrier {
        condvar notLast
        int n, nvalue;

        void init(val) {
                n = nvalue = val
        }

        void wait() {
                n = n -1
                if (n > 0)
                        notLast.wait()
                else
                        notLast.signal()
        }

        void reset() {
                n = nvalue;
        }
}
```

    b. Implement a modified semaphore using monitor. In this semaphore, the wait() can take a priority value. When a signal is made on the semaphore, it wakes up the process that has the highest priority among the ones already waiting. If no process is waiting, the semaphore remembers the signal and lets the next wait() to fall through whatever its priority is.

```
Monitor WeightedSemaphore {
        condvar semNotReady[N]
        int value
        int waitCount[N]

        init(int ival) {
                value = ival
                initialize waitCount array to 0.
        }
```

```
        void wait(int weight) {
                value = value -1
                if (value < 0) {
                        waitCount[weight]++;
                        semNotReady[weight].wait();
                }
        }

        void signal() {
                value = value + 1
                if (value >= 0) {
                        for I = N downto 0:
                                if (waitCount[i] > 0) {
                                        waitCount[i]—
                                        semNotReady[i].signal()
                                        break
                                }
                }
        }
}
```

4. Scheduling problems [**25 points**]
    a. Describe the Stride scheduling algorithm. Be brief and provide the key details.

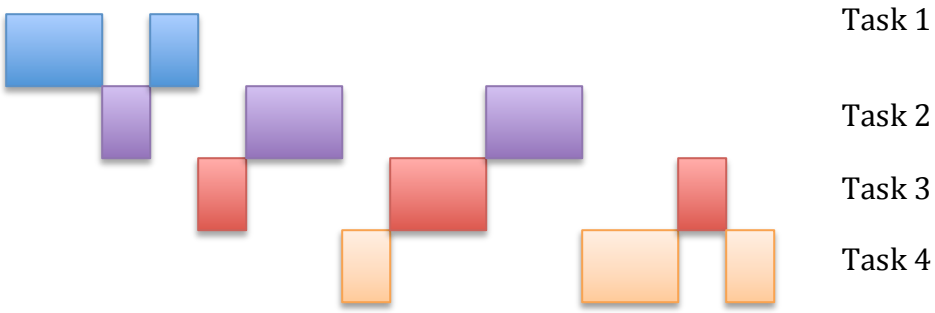
Here is a simple description of stride scheduling.

Each process has a meter value. When a process runs on the CPU, it's meter value is incremented. The rate at which the meter is incremented is equal to 1/bribe, where bribe is the number of tickets the process has. The scheduler at any given time picks the process with the smallest meter value.

    b. Rate monotonic scheduler is asked to schedule a task with deadline 50 and service time 30. Give a second task it cannot schedule but an earliest deadline scheduler can schedule.

Consider another task with a deadline of 75 and service time of 25. This task has lower priority than the first task. The RMA will schedule the first task every 50 time units. In the first 75 units, there will be only 20 free time units. Therefore, RMA cannot schedule the second task. The earliest deadline first scheduler can schedule both tasks. In the first 150 time units, we have 30 + 25 + 30 + 25 + 30 = 140 time units. Therefore, EDF can find a feasible schedule.

c. Schedule the following tasks using a multi-level feedback queue scheduler. There are four levels in the scheduler and the maximum quantum increases from 1 (at the top) to 8 at the bottom. Task 1 (arrival 0, service 3), Task 2 (arrival 2, service 5), Task 3 (arrival 4, service 4), Task 4 (arrival 6, service 4)



McGill

Midterm Exam: ECSE 427/COMP 310 – Operating Systems

Thursday, March 1, 2012
Place: ENGMC 304 & ENGTR 0070; **Time:** 10:30am – 11:30am

EXAMINER: Prof. M. Maheswaran

STUDENT NAME: _____ **McGill ID:** _____

INSTRUCTIONS:

- This is a **CLOSED BOOK** examination.
- Answer **DIRECTLY** on exam paper.
- No **CRIB SHEETS** permitted.
- You are permitted **TRANSLATION** dictionaries **ONLY**.
- **FACULTY STANDARD CALCULATOR** permitted **ONLY**.
- **THIS EXAMINATION PAPER MUST BE RETURNED.**

QUESTION	POINTS
Question 1	/20
Question 2	/20
Question 3	/20
Question 4	/20
Question 5	/20

[1] When **`fork()`** is executed by a process in UNIX, a child process is created. The child process is a clone of the parent in most aspects because it gets a copy of all key data structures. One of them is the file descriptor table. The file descriptor table points to a table containing context information about the file (we refer to this here as File Systems Table). The **`fork()`** does not copy this table, why? Explain what kind of operations would break if **`fork()`** makes a copy of this table.

The File Systems Table maintains two key pieces of information: file offset and reference count. The file offset gives the current location of access in the file (in some operating systems reads and writes have separate pointers). The reference count maintains the number of file descriptor entries that are pointing to a particular File Systems Table entry. If fork() copies this table, parent process' output will be overwritten by the child process' output or vice-versa. That is we cannot get the "append" behavior we expect.

The Data and BSS segments contain data. Why do we have two different segments for data? What are the disadvantages of merging them into one segment for the sake of simplification?

The BSS is for uninitialized data whereas the Data segment is for initialized data. By having two different segments, different strategies can be used to compress the on-disk image of the segments. For instance, if BSS includes a global uninitialized array, the BSS size would not be as big as the array. When the BSS segment is loaded, the loader will setup the memory map accordingly.

If Data and BSS are combined, the file image will be larger because uninitialized data will be taking up space. Program startup times will increase and disk space consumption will be higher.

What are the advantages of dynamic libraries over static libraries?

With dynamic libraries, the objects (shared objects) are loaded only when needed. The executable file is incrementally loaded as the execution proceeds. Portions of the executable (that are stored in user created libraries) or system library code that are not needed by a particular run are not loaded.

Also, because the shared objects are loaded at run-time, with dynamic libraries it is possible to interpose another version of the library that wraps a preexisting library (function call signature has to be same). By securely exploiting this feature, at run time, program behavior can be changed.

[2] What is the difference between an interrupt and a trap?

An interrupt is caused by an external event. A trap is generated by the program activity. For example, floating point overflow caused by divide by zero can trigger the "floating point exception" trap.

What is the biggest advantage of implementing threads in user space? What is the biggest disadvantage?

The advantage of implementing the threads in user space is that it is efficient. No context switch associated with thread switch. If important functionality such as networking can be carried out by user-level libraries, then user space threads can give a significant performance boost. However, most OSes by default provide important functionality through system call interfaces.

The disadvantage is that a system call that blocks a thread ends up blocking the whole application unless special care is taken to create a different user-level wrapper that calls the underlying system call in a non blocking manner and implements call waiting inside the threading library at the user level.

A multi-threaded process forks and creates a child process. Soon after creating the child process, the parent and child processes run into a deadlock. Explain how this could have happened? If such a deadlock cannot happen, explain why it cannot.

Consider a hypothetical scan-print application. It has two threads one to access the scanner to scan a document and another to access the printer. A process to get the full workflow (scan and print), needs to hold the scanner and also hold the printer. Suppose we fork a child process to multi-task (that is to keep the scanner busy while the printer is printing out the first document). The parent and child could run into a deadlock if the parent holds the scanner (remember the scanner and printer are controlled by two different threads) and the child holds the printer.

This deadlock scenario happens when the thread forking model allows the same number of threads as the parent in the child. If the child has only thread no matter how many threads the parent has, then this deadlock would not happen.

Some shell commands are run in separate processes and other are run inside the shell's process. Explain the distinction between the two types of commands that require different implementation strategies.

Some shell commands such as "cd" change the state of the shell. In particular, the cd command changed the current working directory. Any command that interacts with the file system will be impacted by the change made by cd. If cd were to run in a separate process, the change made by running the command would not change the state of the shell. That is subsequent commands would not be impacted by cd.

[3] There is a memory block of 128 KB that is going to be allocated by an OS for requests that arrive at run-time. Suppose the request sequence is 17KB, 9KB, 5KB, and 3KB and the whole block is free at the beginning of the sequence. Show the allocation if the binary buddy algorithm is used for the allocation process. In your diagram show where the memory is allocated and how much memory is used from each "block" that is allocated. For the whole sequence, find the memory wasted due to internal fragmentation (total amount of memory left unused inside each block). Suppose the buddy algorithm used the Fibonacci sequence to break the block, show the allocations and compute the internal fragmentation. The Fibonacci sequence is given by $F_0 = 0$, $F_1 = 1$, $F_n = F_{n-1} + F_{n-2}$.

With binary buddy, the following allocation will take place.

128 -> [64, 64] -> [[32 (allocated to 17), 32], 64]
- ➔ [[32 (allocated to 17), [16 (allocated to 9), 16]], 64]
- ➔ [[32 (allocated to 17), [16 (allocated to 9), [8 (allocated to 5), 8]]], 64]
- ➔ [[32 (allocated to 17), [16 (allocated to 9), [8 (allocated to 5), [4 (allocated to 3), 4]]]], 64]

Total internal fragmentation = (32 – 17) + (16 – 9) + (8 – 5) + (4 -3)
= 26KB

With Fibonacci series, the following sizes are available: 0, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144 (bigger than 128).

So we can split 128 KB as follows using Fibonacci numbers

128 -> [89, 39] – lets ignore the 39 for now. Fibonacci can be used to split it as well. Either way, it does not change the results for internal fragmentation.

89 -> [55, 34] -> [55, [21 (allocated to 17), 13]]
- ➔ [55, [21 (allocated to 17), 13 (allocated to 9)]]
- ➔ [[34, 21], [21 (allocated to 17), 13 (allocated to 9)]]
- ➔ [[34, [13, 8]], [21 (allocated to 17), 13 (allocated to 9)]]
- ➔ [[34, [13, [5 (allocated to 5), 3 (allocated to 3)]]], [21 (allocated to 17), 13 (allocated to 9)]]

Total internal fragmentation = (21 – 17) + (13 – 9) + 0 + 0

[4] You need to develop a monitor-based solution for a ***student consultation office problem***. In this problem, students visit their student advisor to get advise. The advisor has grouped the students in two groups. There are n chairs for the students to wait in the office, if the advisor is busy. If the advisor is not busy, an arriving student starts getting the advise immediately. If the advisor is busy, student takes a chair if one is free. If all chairs are taken, student leaves without advise. Also, if a student belonging to one group is taking advise, the advisor does not want students belonging to the other group in the office. The students belonging to the other group can come into the office only after all of the students belonging to the current group have left the office. Develop a solution based on monitors. Identify the most important problem you can face with the solution you propose. Suggest an approach for solving it (No need to implement it – just describe the approach in words).

```
Monitor Student_Consultation {

   int groupType;
   int advisorState = {BUSY, DONE, FREE}
   condvar advisorFree;
   List chairs;

   int requestAdvisor(int mygroup) {
         if (advisorState = FREE) {
               advisorState = BUSY
               groupType = mygroup;
               return TRUE;
         }
         if (!((groupType == mygroup) && ((mychair = chairs.getAChair()) !=
NULL)))
               return FALSE;
         while (mychair != chairs.TopChair())
               advisorFree.wait();
         advisorState = BUSY;
```

```
    }
    releaseAdvisor() {
            if (chairs.empty())
                    advisorState = FREE;
            else {
                    advisorState = DONE;
                    advisorFree.broadcast();
    }

    init () {
            chairs.init(n);
            advisorState = FREE;
    }


}

// Assume a List structure that does not own its objects.
// TopChair would always be distinct. It does not depend on the slot - it
// depends on the actual object
```

[5] What are the necessary and sufficient conditions for deadlocks to occur?

(a) Hold and wait
(b) Circular wait
(c) Mutual exclusion
(d) No preemption

Other equivalent statements of the above four conditions are acceptable answers.

Which of these conditions are easy to prevent in a system where deadlocks should not occur by design? Briefly explain your answer.

Circular wait is the easiest to break by design. We could create an ordering which should be respected when seeking more than one resource. This way deadlocks can be easily prevented.

Hold and wait is another condition that can be prevented as well. However, this means the process needs to retry for the resource, which it once held. Implementing this strategy can increase the complexity of the program.

Mutual exclusion is a correctness issue. Some activities cannot be performed by more than one process in a simultaneous manner. For such activities mutual exclusion is essential.

There is a computer system with 250KB of main memory. A Bankers algorithm is in place to avoid deadlocks from happening as the memory allocation proceeds. Suppose there are four processes P1, P2, P3, P4 already admitted with the following amount of memory allocated to them: 90KB, 30KB, 50KB, and 20KB. What is the maximum value of total claim the four processes can have if the allocation is in safe (not very safe) state?

Total maximum claim is 830KB.

Total allocation at this point is 190KB. Memory available is 60KB.

Allocating this memory to P1 is the best option because it frees the most amount of memory when P1 completes. So the best sequence is P1 – P3 – P2 – P4.

P1 claim = 60 + 90 = 150KB
P3 claim = 150 + 50 = 200KB
P2 claim = 200 + 30 = 230KB
P4 claim = 230 + 20 = 250KB

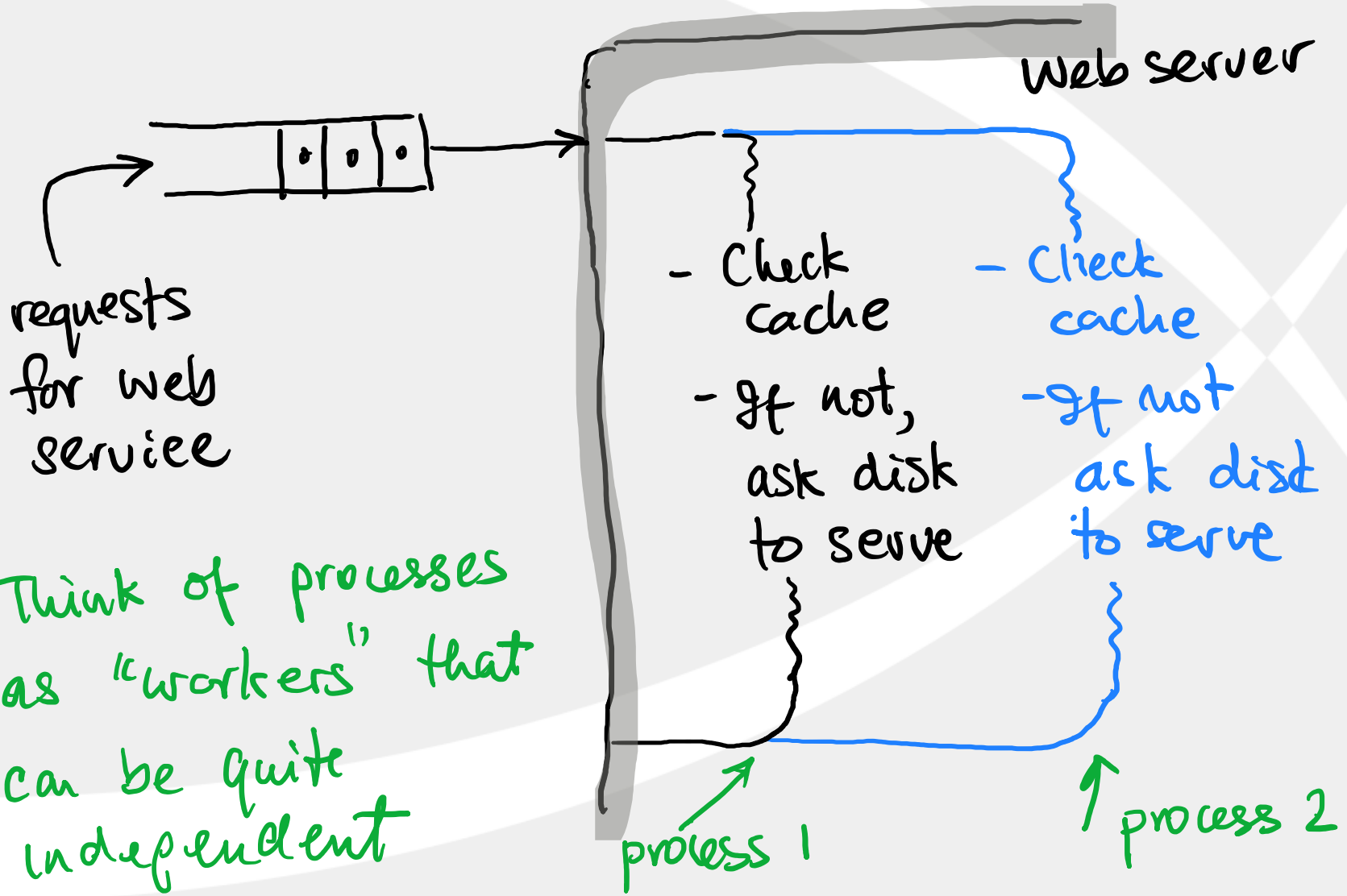
Total claim = 830KB

Processes and Threads

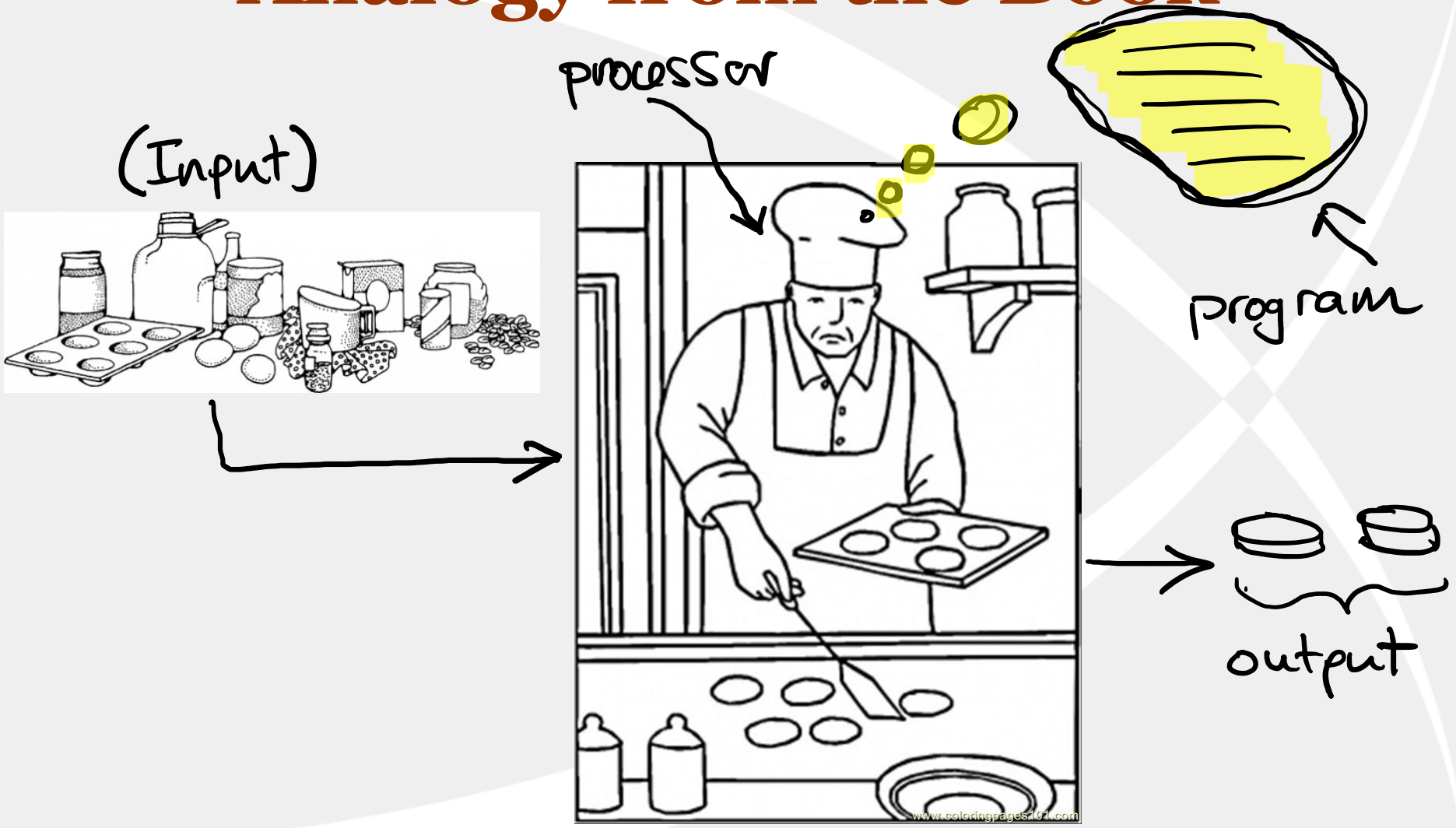
What is a *process*?

- 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

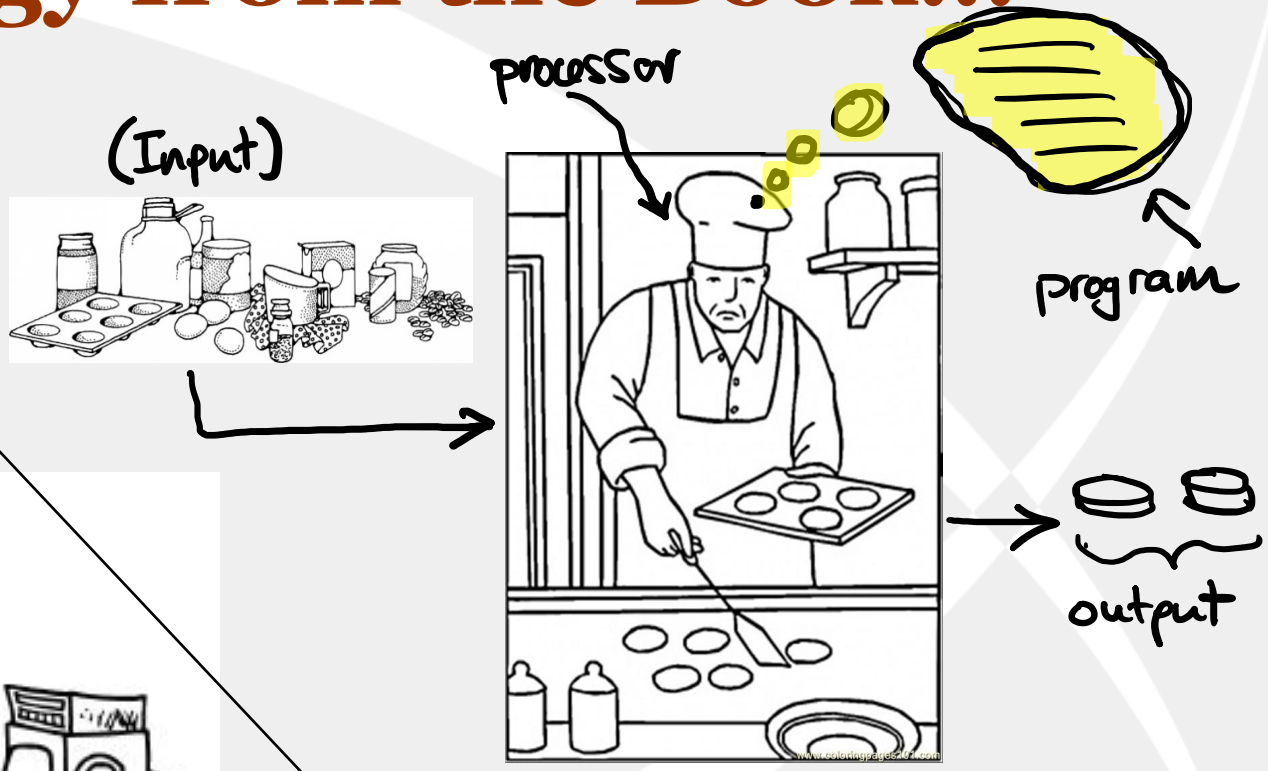


Analogy from the Book

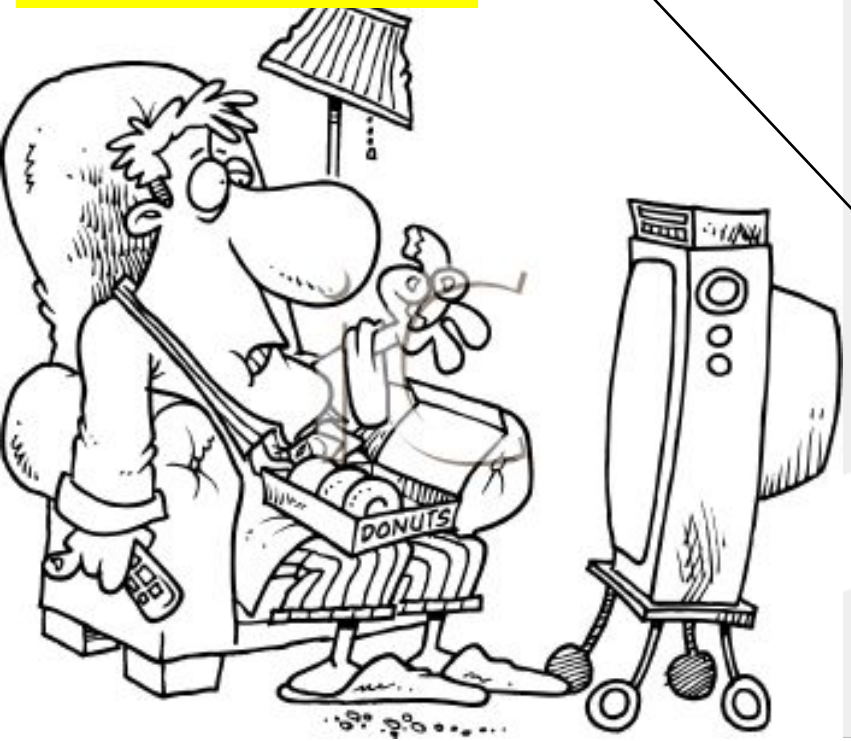


"Process" is the activity of baking the cookies

Analogy from the Book...



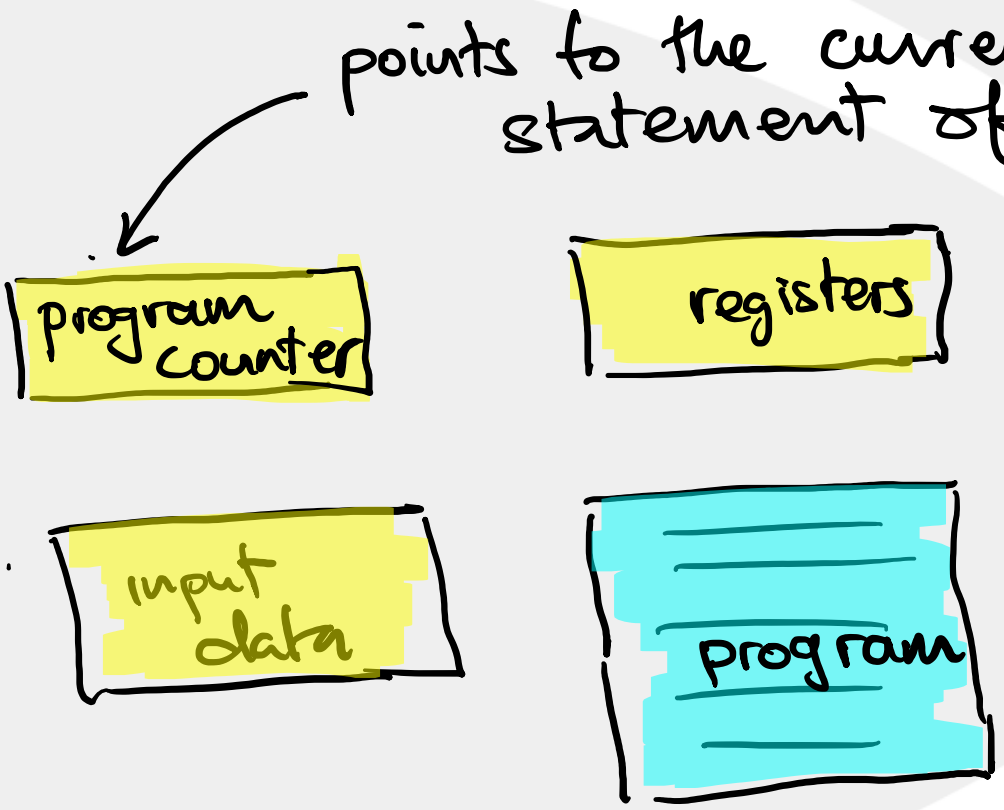
"Process" is the activity of watching football



"Process" is the activity of baking the cookies

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

Key Elements of a Process



some key elements missing... what are they?

Multiprogramming

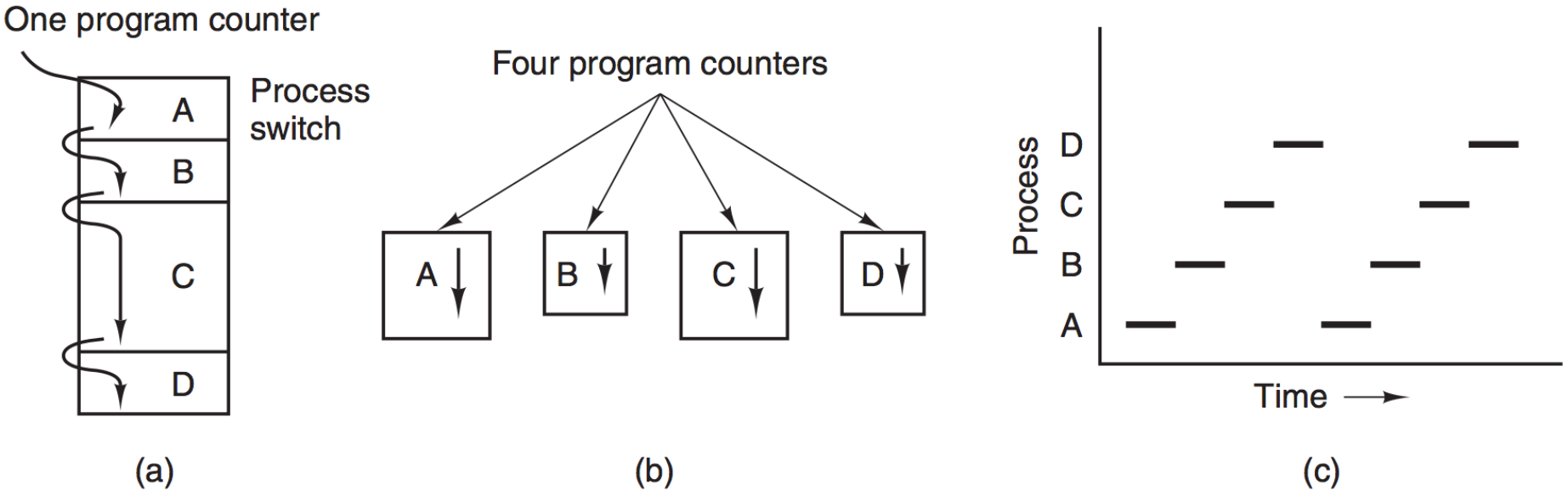


Figure 2-1. (a) Multiprogramming four programs. (b) Conceptual model of four independent, sequential processes. (c) Only one program is active at once.

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

multiprograming

Program X

Program Y

Program Z

No need for the process concept

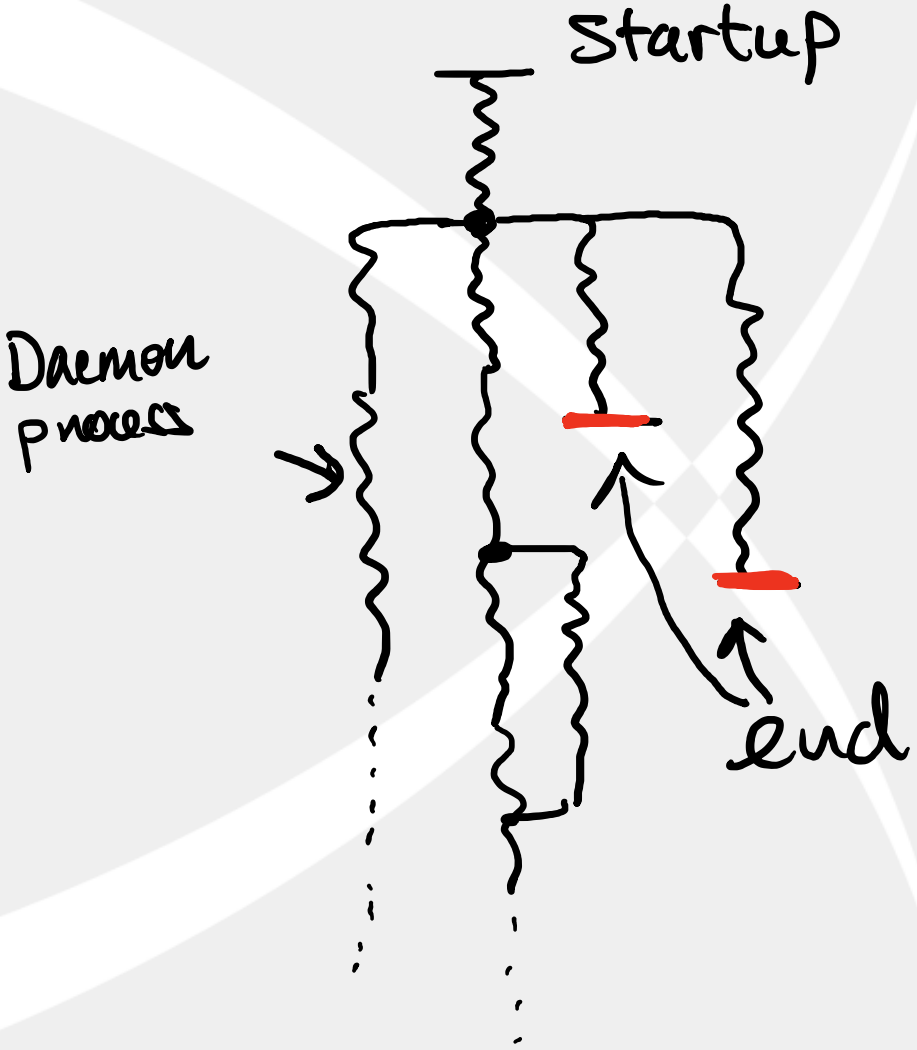
switch between different programs that are elisible to run.

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

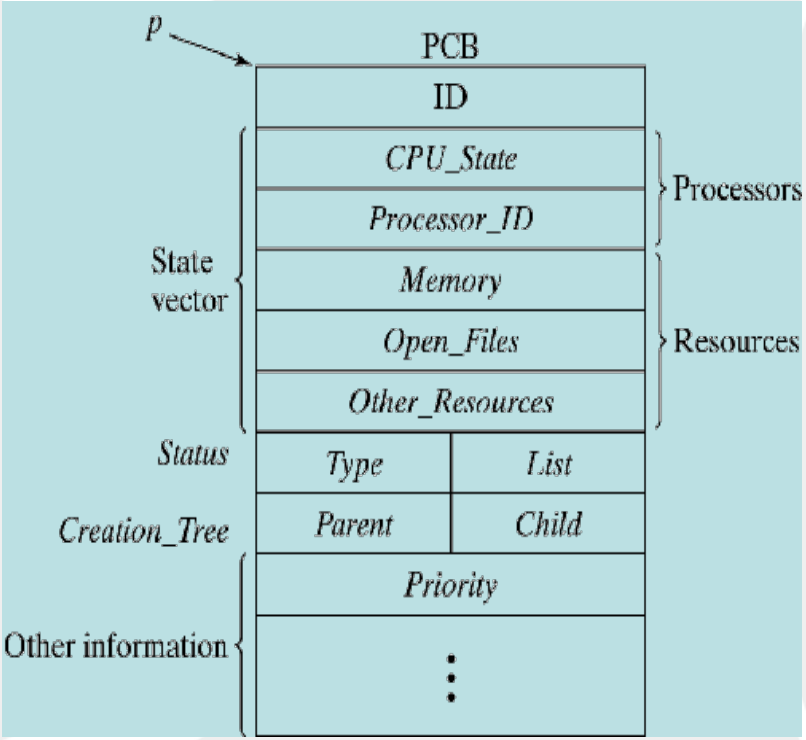
Process Creation

- 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



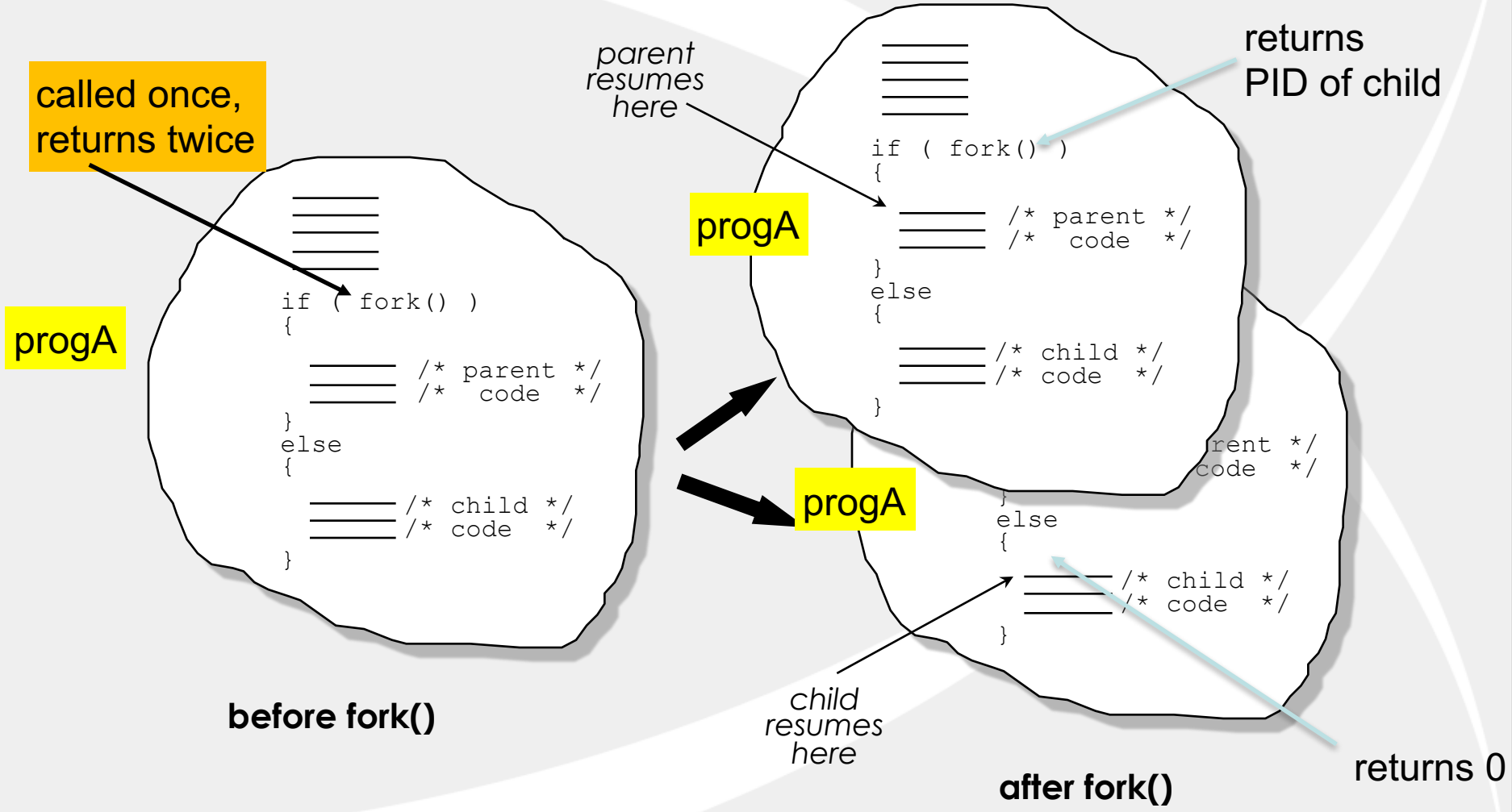
Lifecycle: Create process

- Two ways of creating a new process:
  - *Build one from scratch*: ← Windows approach
    - 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)
  - *Clone an existing one*: ← UNIX approach
    - 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)

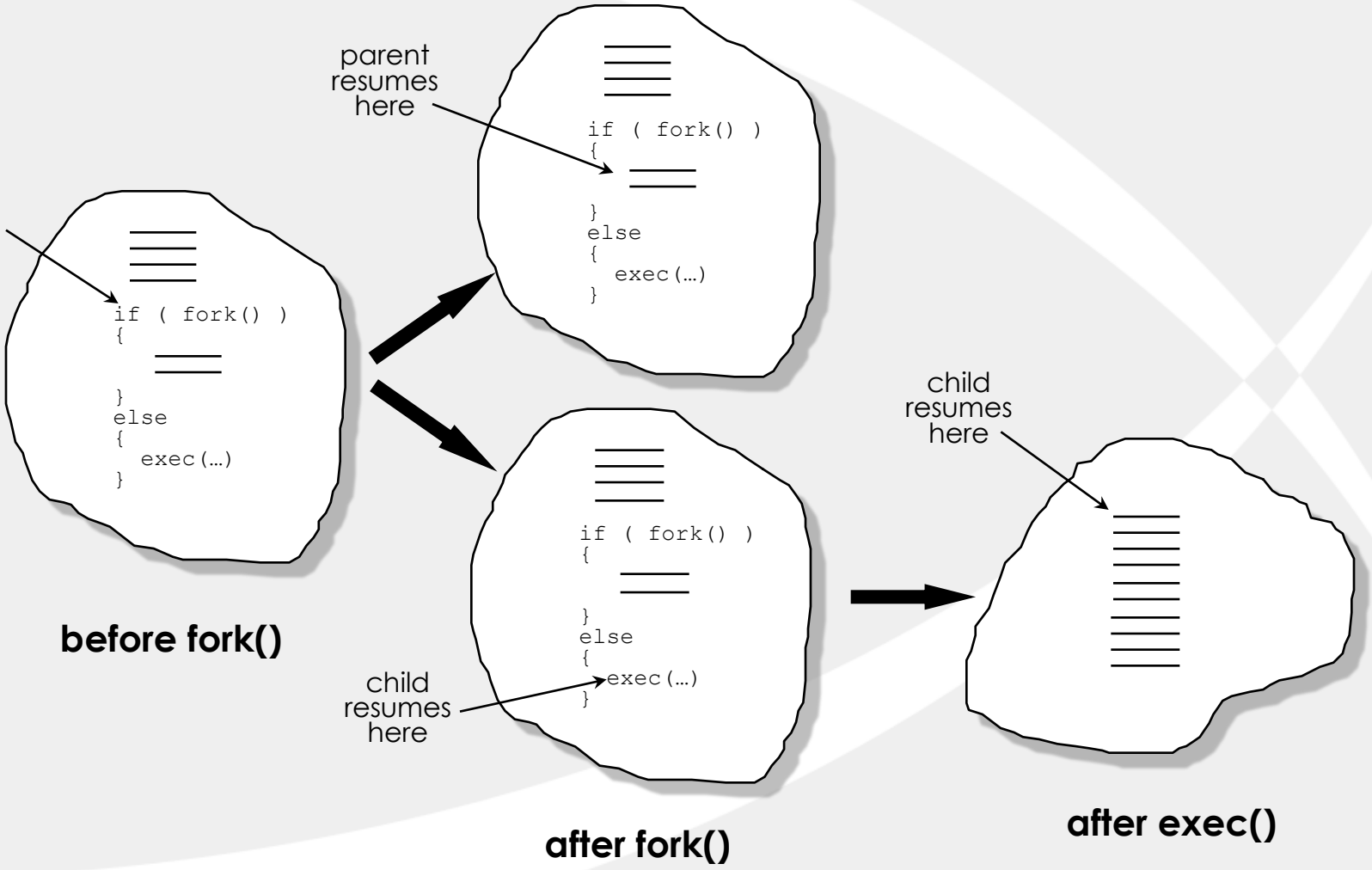
UNIX process creation

- 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

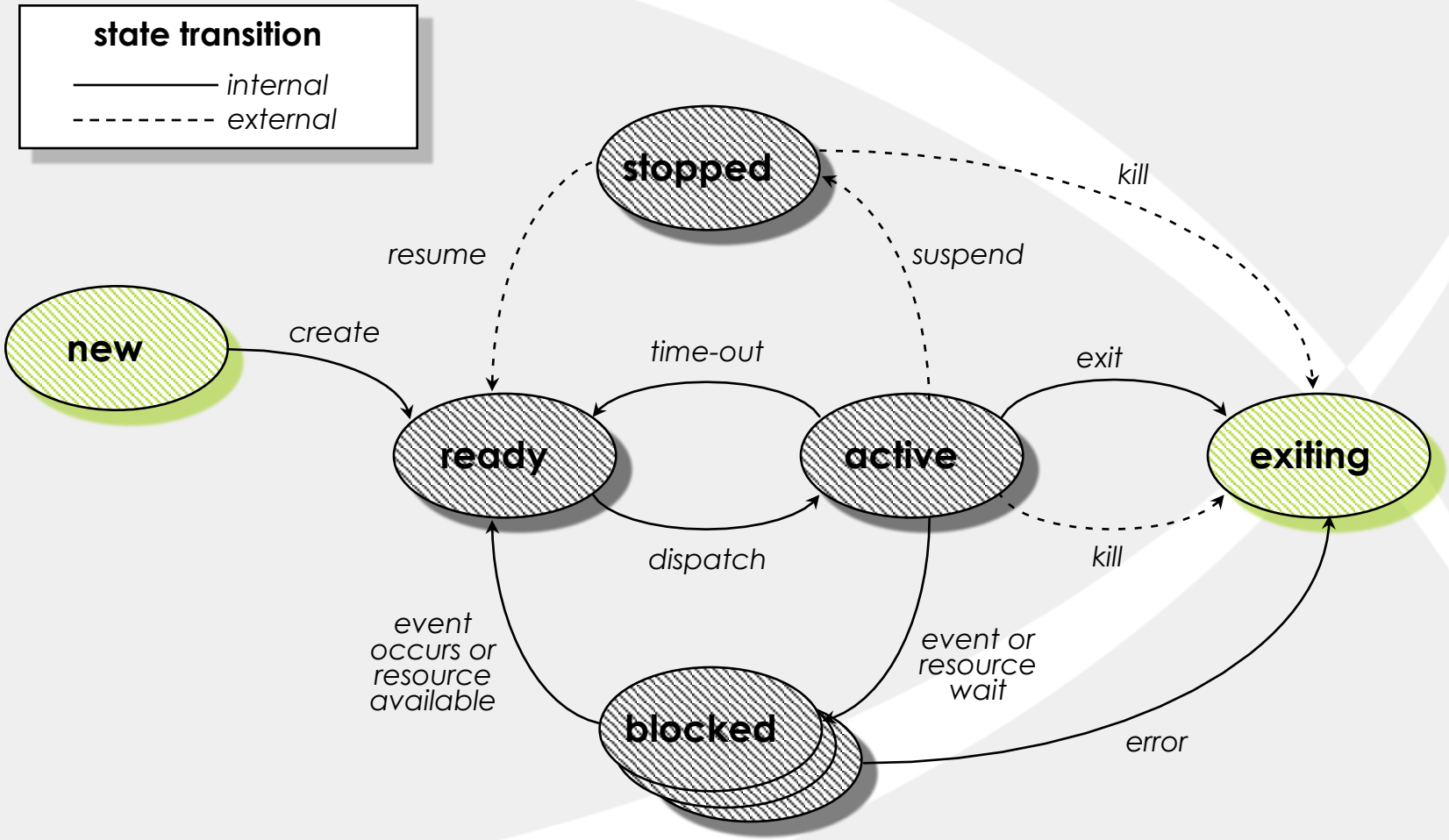
- What is the output of the following simple C program?

```
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

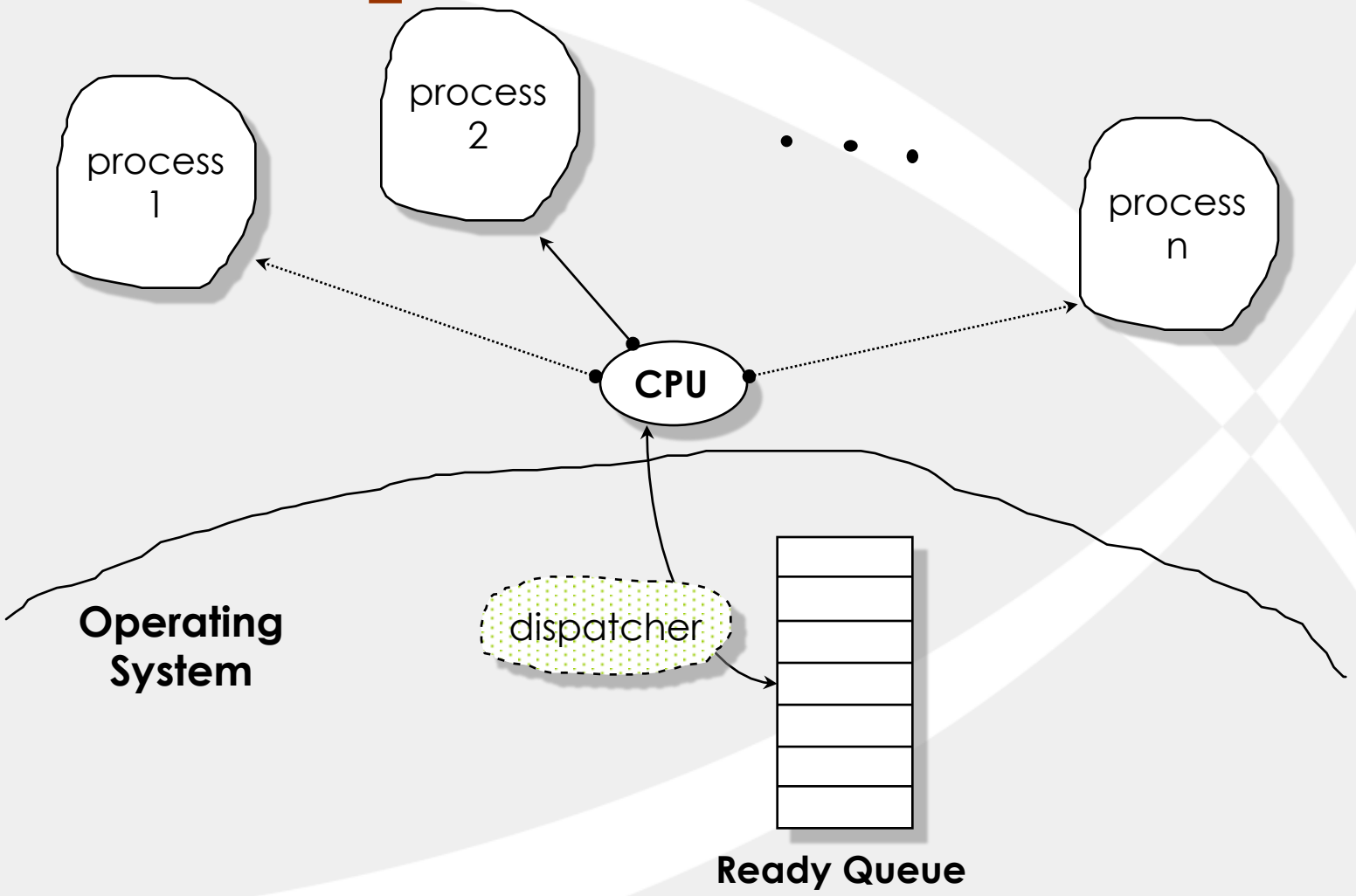
```
while (1) {
    printf("Shell > ")
    getline(line)
    if (strlen(line) > 1) {
        if (fork() == 0) {
            exec(line)
        }
        wait(child)
    }
}
```

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?

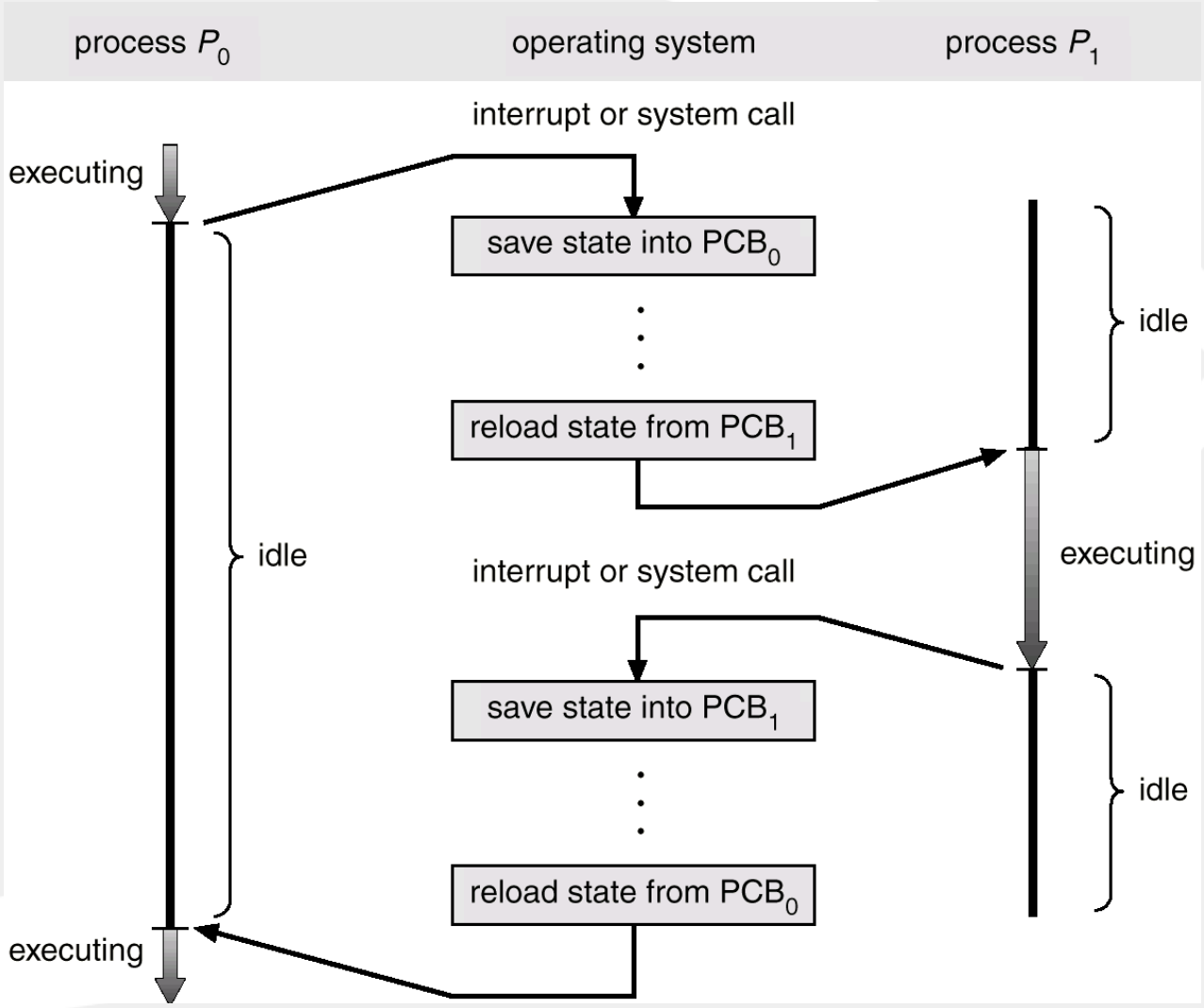
  - Don't save memory; trust next process.

    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 C

Original application..

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

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

Process A	Process B	concurrent access
`A = 1;`	`B = 2;`	*does not matter*
`A = B + 1;`	`B = B * 2;`	*important!*

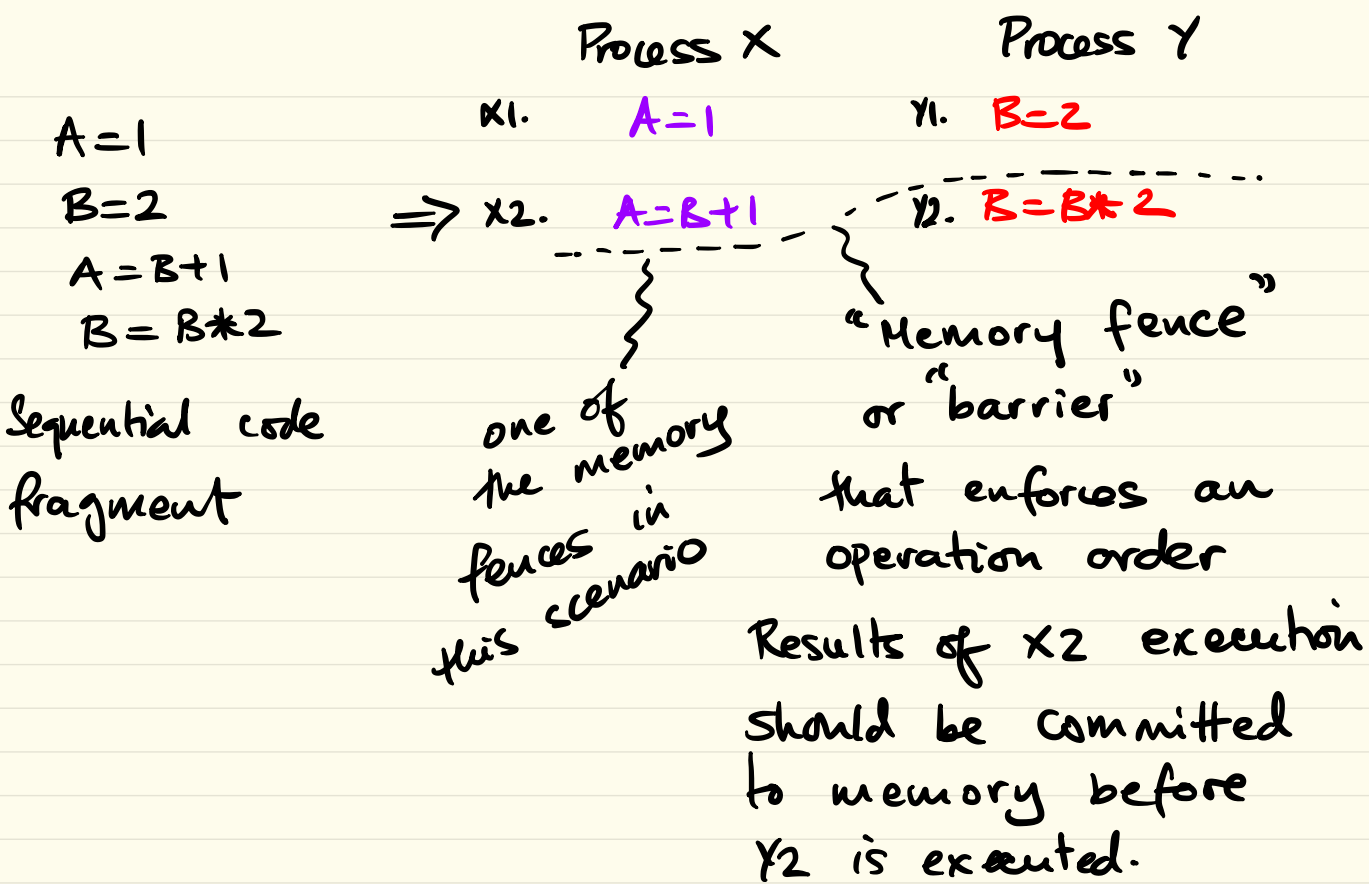
Consider a sequential program fragment

```
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
Process Y
A=1
B=2
A=B+1
B=B*2
Sequential code fragment
X1. A=1
X2. A=B+1
Y1. B=2
Y2. B=B*2
one of the memory fences in this scenario
"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)

A & B are updating a
drawing and B depositing.

Initial balance = $100

Process A
R1 <- $100
R2 <- $50
R1 <- $100 - $50
balance <- R1 = $50
store R1, balance

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 ← $100 (balance)
// substract the amount
R1 ← R1 – $50
// Update the balance
R1 → balance

Deposit Process

// Retrieve balance in R1
R1 ← $100 (balance)
// Add. the dep. amount
R1 ← R1 + $50
// update the balance
R1 → balance.

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

The following execution sequence gives $50.

R1 ← $100 (balance)

R1 ← R1 − $50

R1 ← $100 (balance)

R1 ← R1 + $50

R1 → balance ← $150 in balance.

R1 → balance ← $50 is written into balance by the resumed deposit process.

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

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
Critical section
        load R1, balance
        load R2, amount (-ve for withdrawal)
        add R1, R2
        store R1, balance
        close account
        display account info
```

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

Requirements of a 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

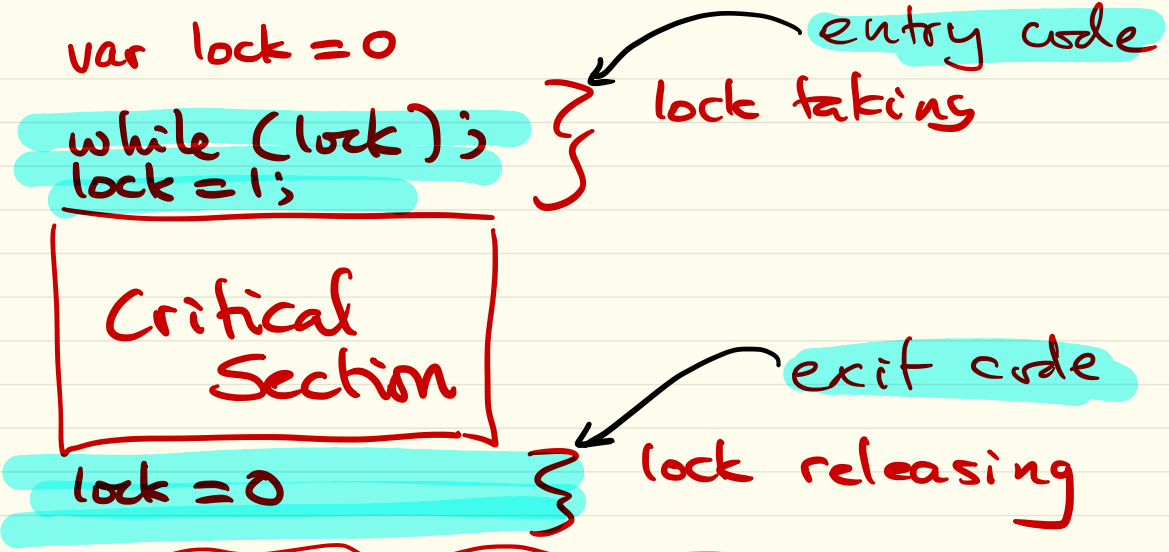
- Critical region execution should be *atomic* i.e., its runs “all or none”
- Should not be interrupted in the middle
- For efficiency, we need to minimize the length of the critical sections
- Most inefficient scenario
  - Whole program is a critical section – no multiprogramming!

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??



lock = 0

while (lock); ~ empty statement

≡

lock = 0

The simple idea of using a "lock" variable does not wok 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                          Process 1


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

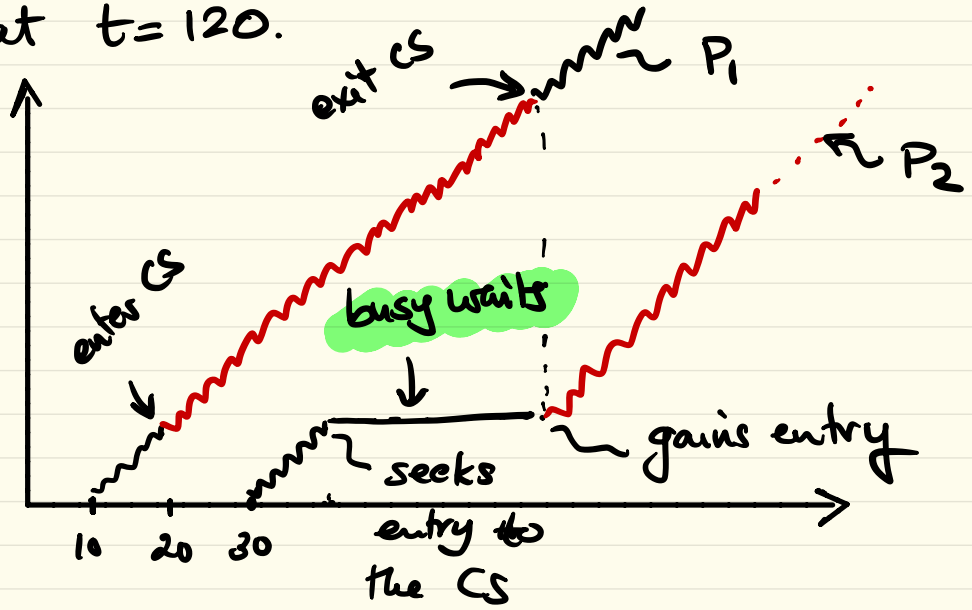
Strict Alternation

- 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 10 s.
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?

Lets make the following assumptions:

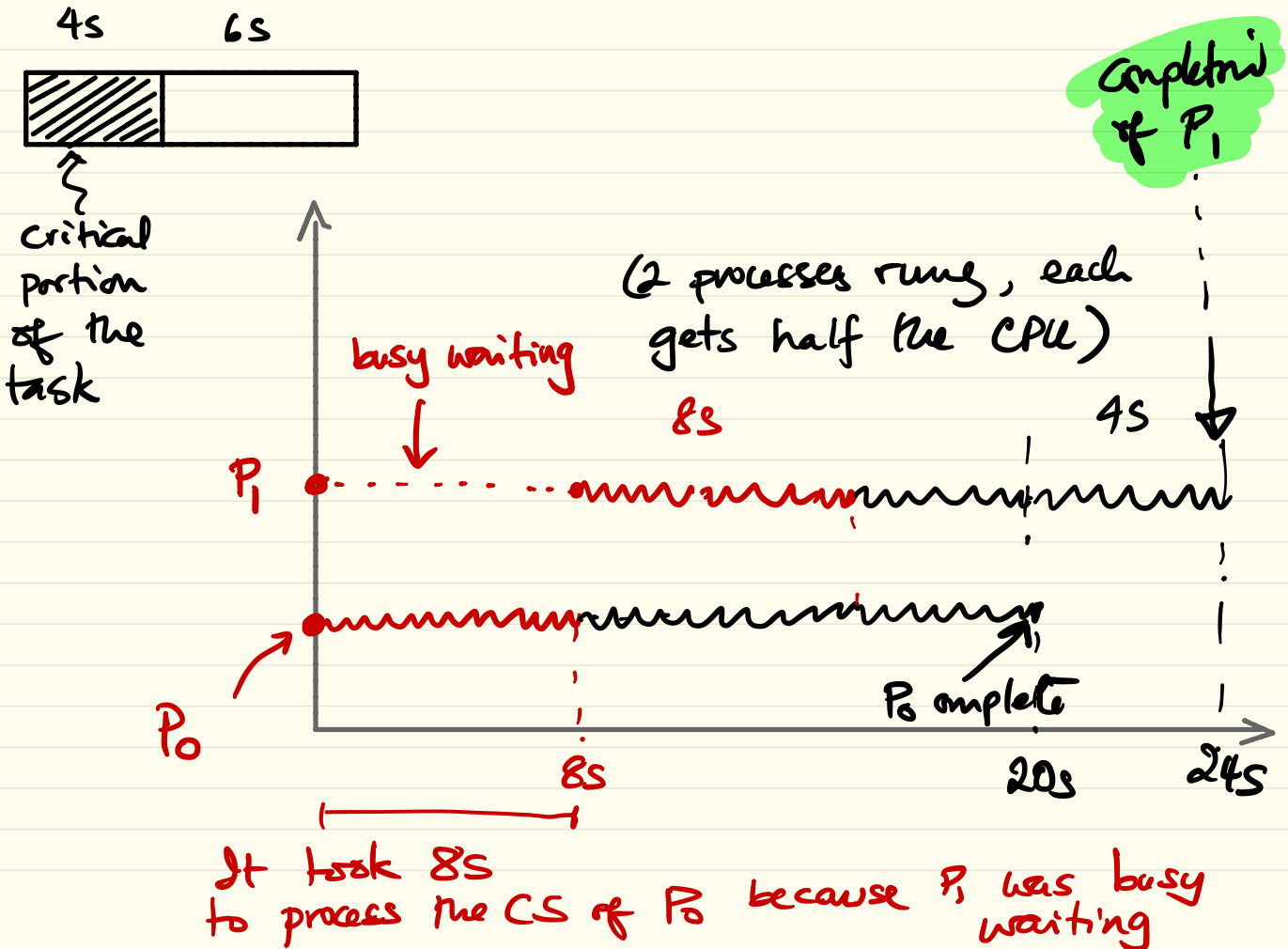
- single CPU, single core.
- Ideal time sharing, no other tasks besides these two tasks.

Simple argument: 2 tasks 20s (total workload)

So we need at least 20seconds.

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.


4s
6s
critical portion of the task
Completion of P_1
(2 processes running, each gets half the CPU)
busy waiting
8s
4s
P_1
P_0
P_0 completes
8s
20s
24s
It took 8s to process the CS of P_0 because P_1 was busy waiting

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

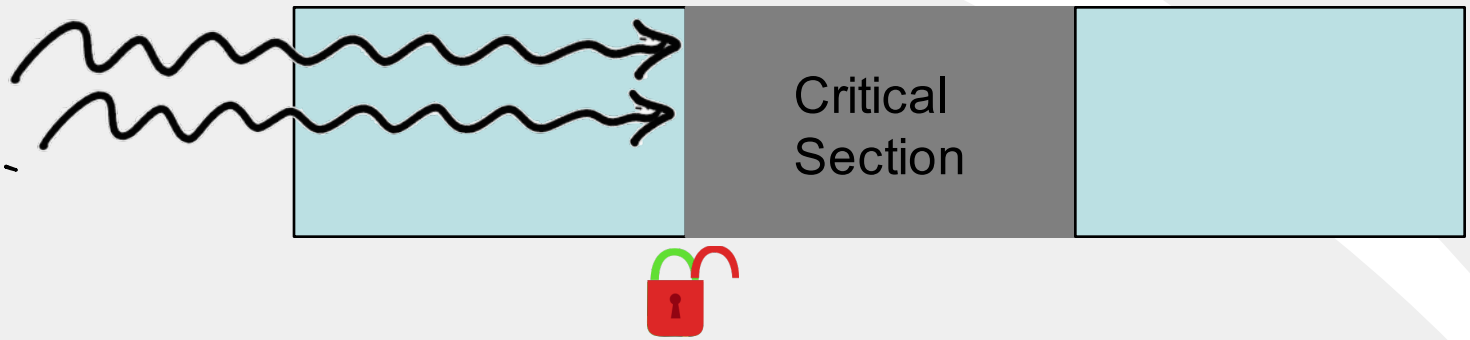
What can you say about the use of busy waiting? Would you recommend its use in certain situations?

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.

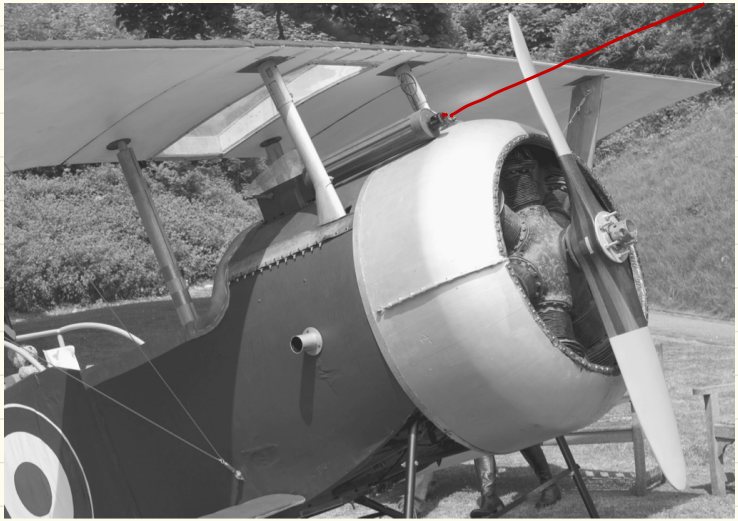
Locks – An Illutration



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

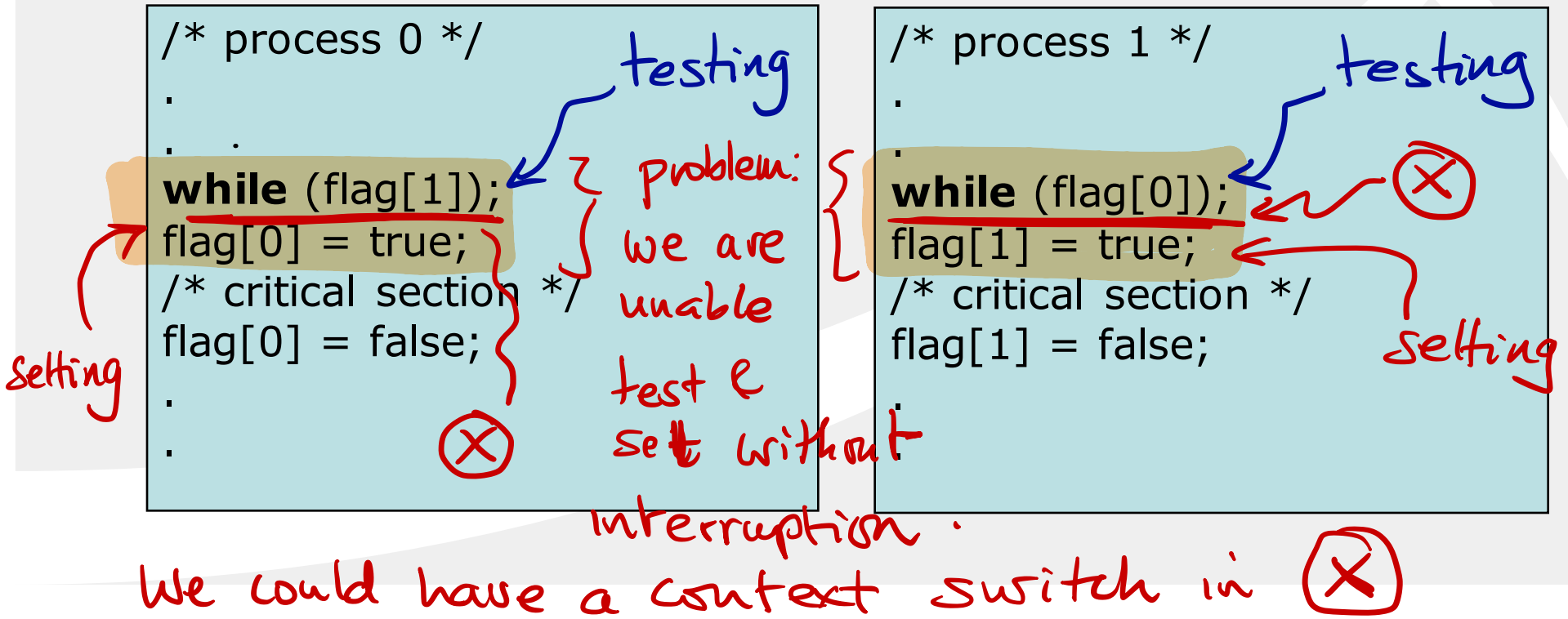
- The simplest mutual exclusion strategy is taking turns as considered earlier

```
/* 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??



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?

One way of attempting to solve deadlock is by "being nice". Where we give up the demand and try to re-grab.

```
/* 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;
.
.
```

This can lead to live lock 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?

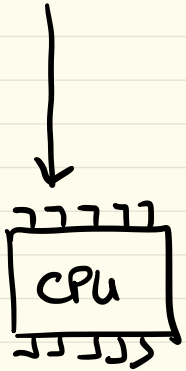
- Dekker's and Peterson's algorithms are pure software solutions
- Can hardware provide any help?
  - make the solution more efficient
  - make it scalable to more processes?
- Yes!
- Current microprocessors have hardware instructions supporting mutual exclusion

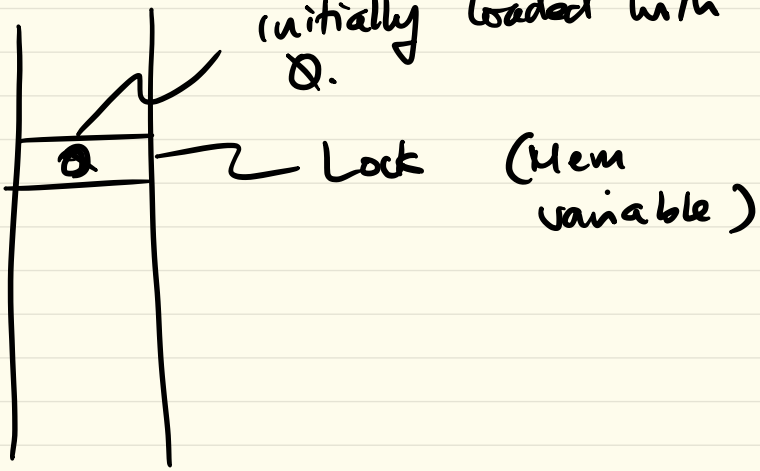
Recall from a earlier discussion that a simple idea of using a "lock" variable, could have worked for providing a critical section if we could do the Testing and Setting atomically. That is once the value is tested and ascertained to contain a certain value we perform a certain modification on the value,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

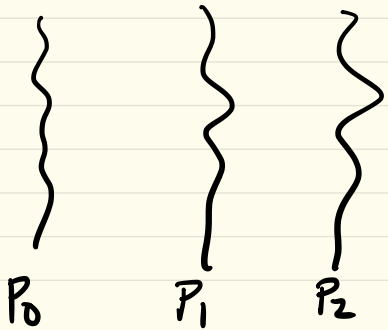
This is the basis of hardware implementations. There are various forms with subtle variations in their capabilities.

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

TRL (R, Lock)







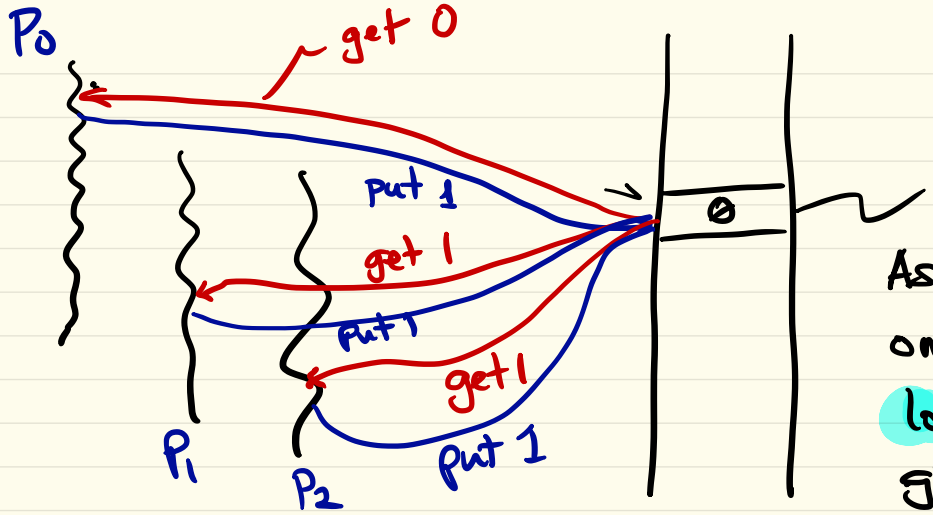
Suppose P_0, P_1, and P_2 execute TRL (R, L) simultaneously, we claim only one process will get 0 in its register.

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 one 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.

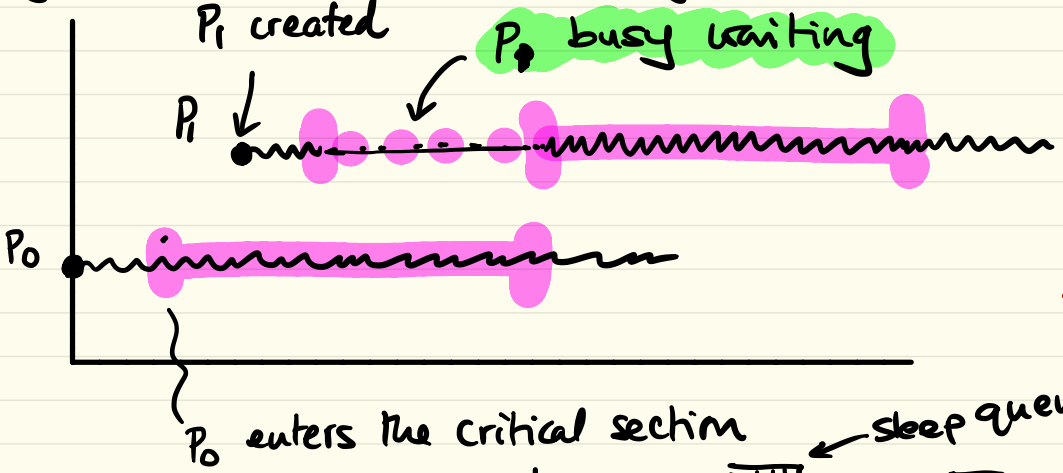
P_1 is unable to get into the critical section X because P_0 is still holding it. P_0 is unable release it because it is unable to run on the CPU.

Sleep/Wakeup Approach

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

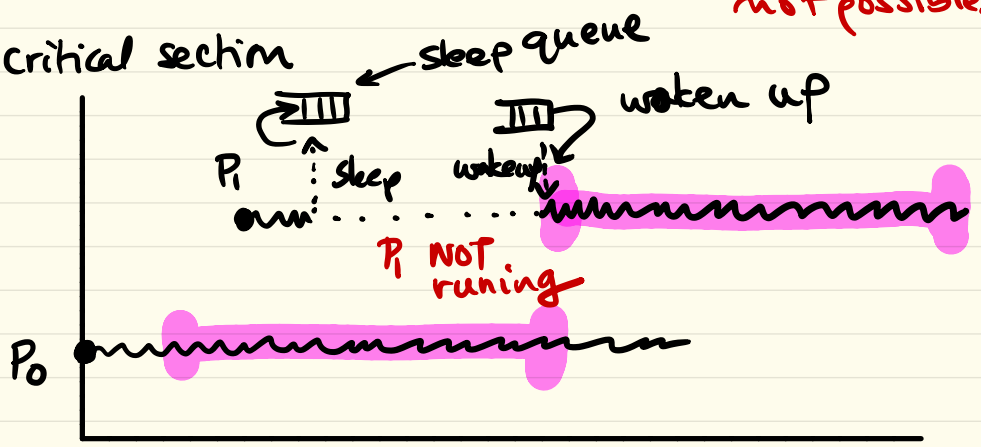
OS support is needed to implement process "sleep" and "wakeup". Most OSes provide semaphores as part of their system call offering.

Lets contrast busy waiting and sleep/wakeup to get a better understanding.



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

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



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

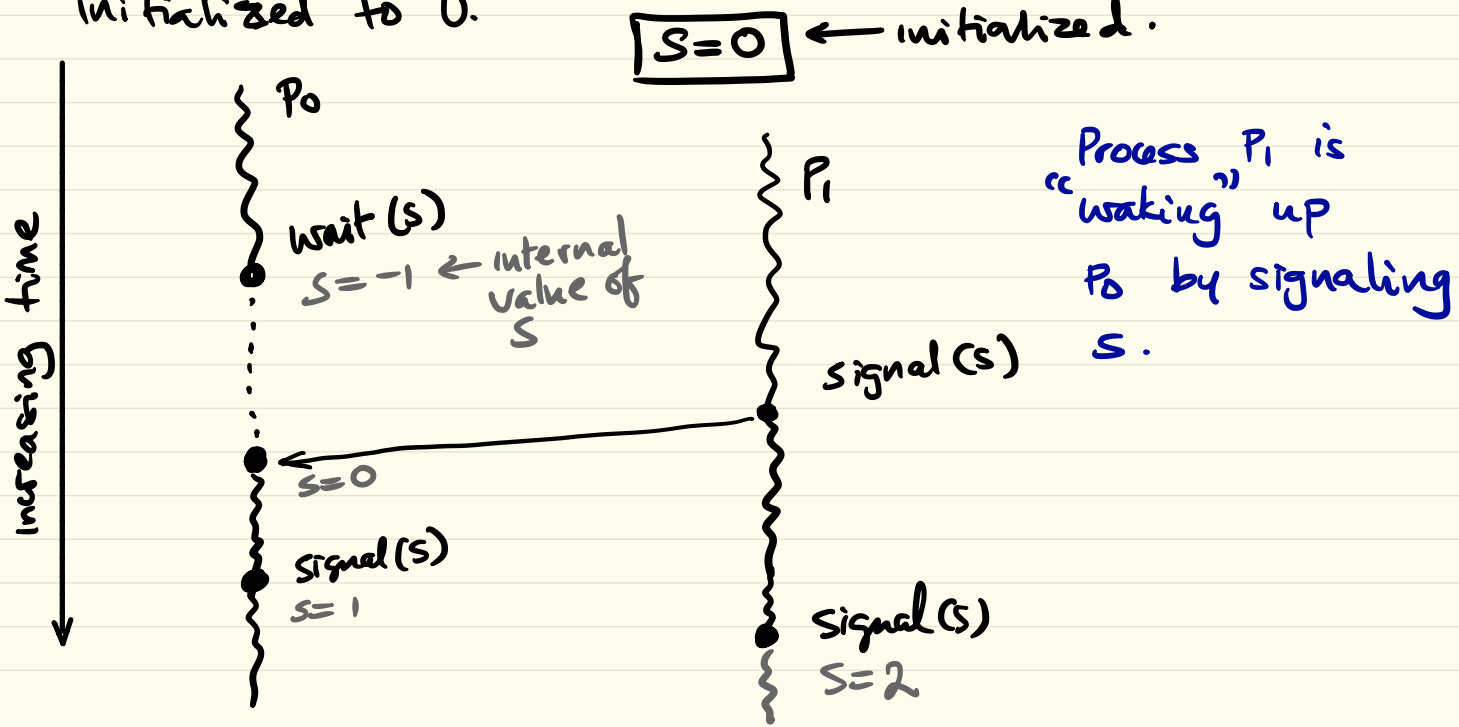
- Wait and Signal primitives are assumed to be atomic
  - they cannot be interrupted and treated as an indivisible step
- A queue is used to hold the processes waiting on a semaphore
- How are the processes removed from this queue?
  - FIFO: process blocked longest should be released next – ***strong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

Type of sema

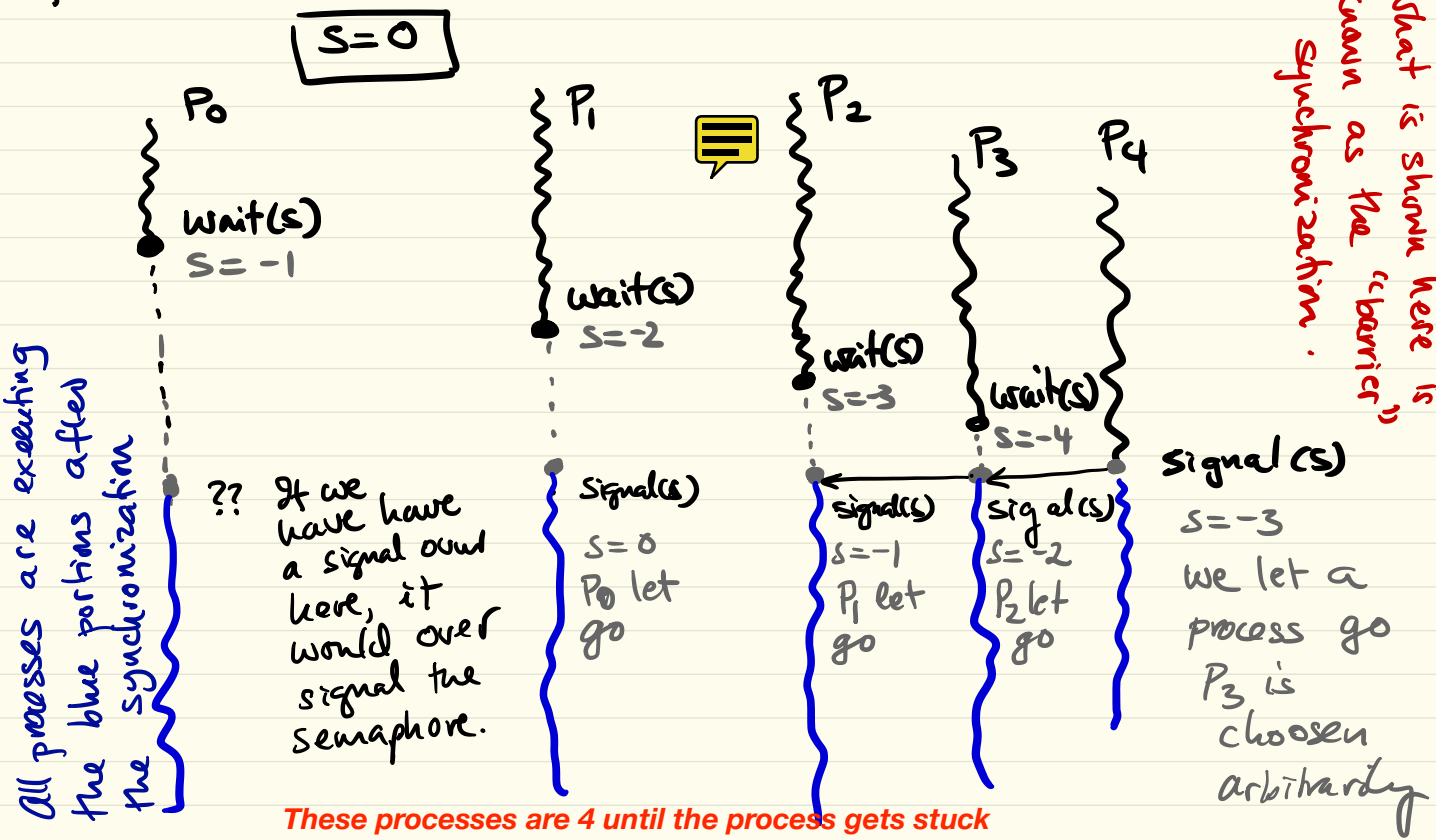
- Semaphores are usuall
  ways:
  - **binary**—is a semaphore wi
    of 0 or 1
    - Or a value of FALSE o RUE if you prefer
    - Initialized to 1 for mutex applications
  - **counting**—is a semaphore with an integer value ranging between 0 and an arbitrarily large number - initial value might represent 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.

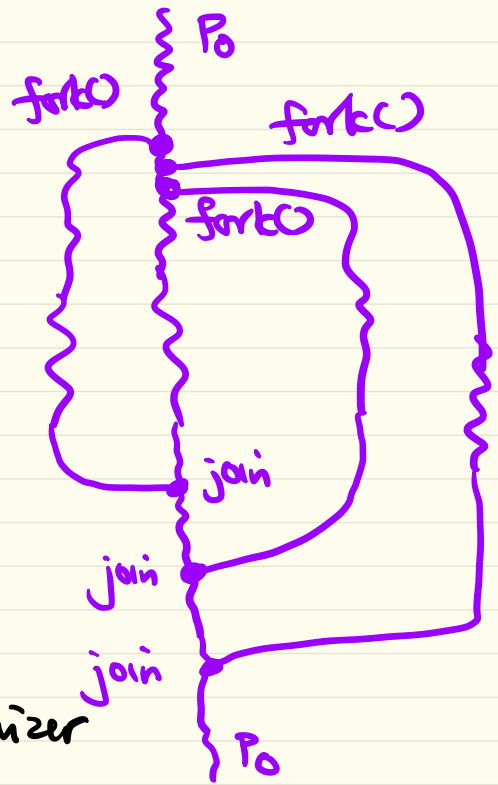


Lets consider another scenario. Here we have 5 processes. We want to synchronize their executions.



These processes are 4 until the process gets stuck because of the memory aspect of semaphores.

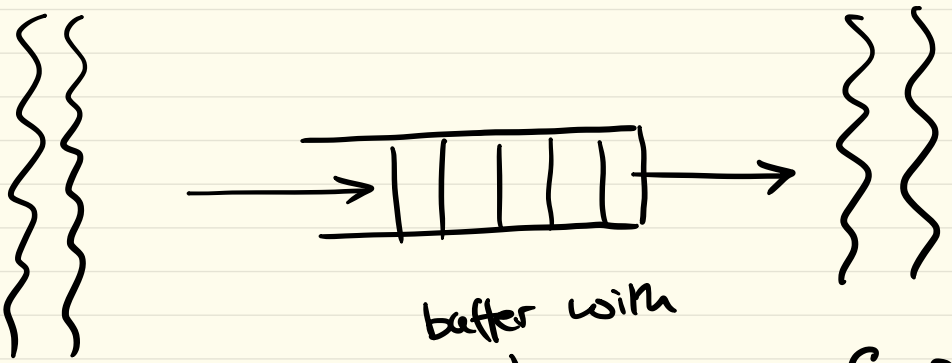
As another example consider a fork-join graph like what is shown next.



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

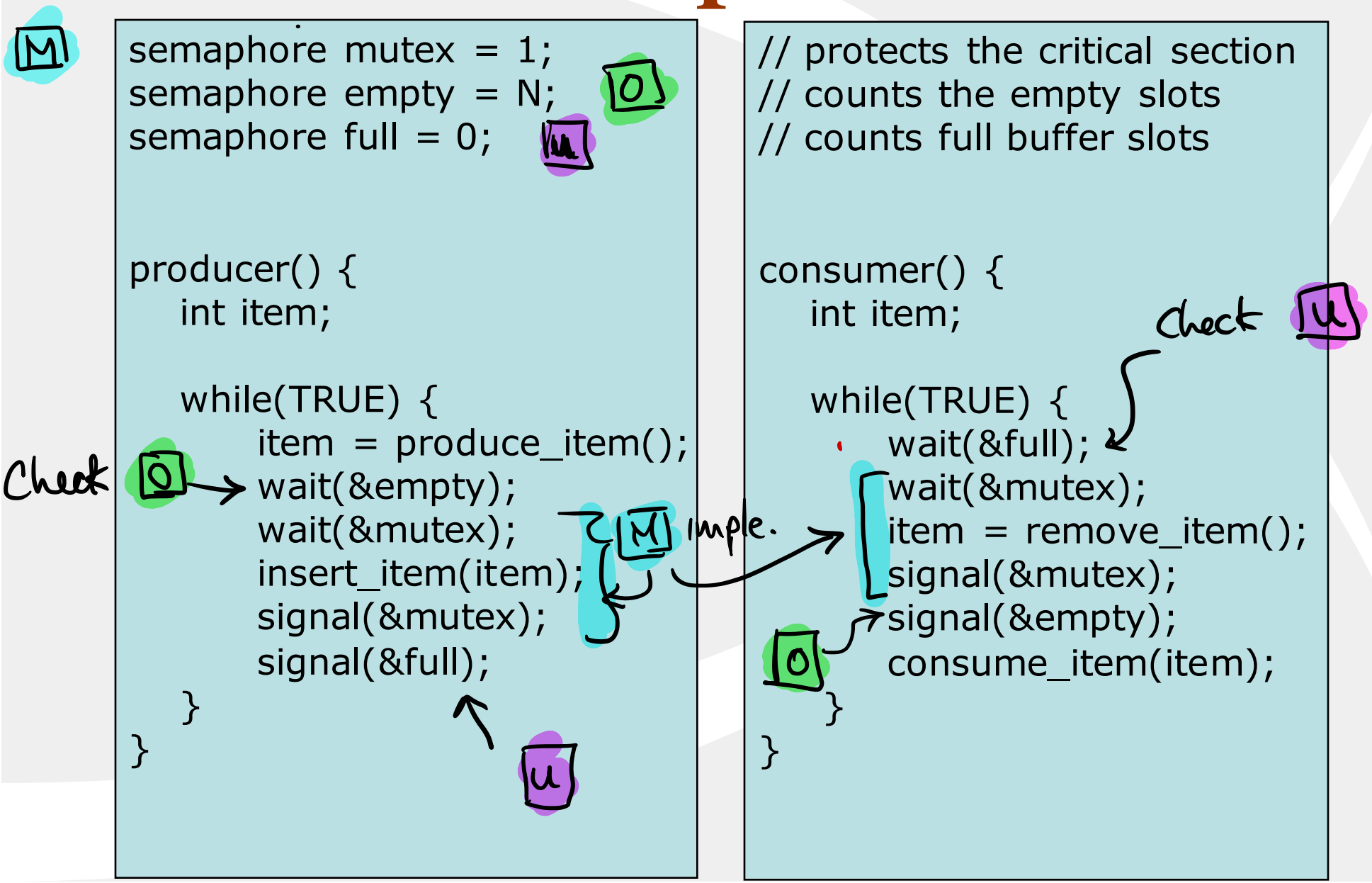
So we have three synchronization requirements.

– 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 any one process modifying the shared buffer structure for race-free operation.

Producer-Consumer: Semaphores



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

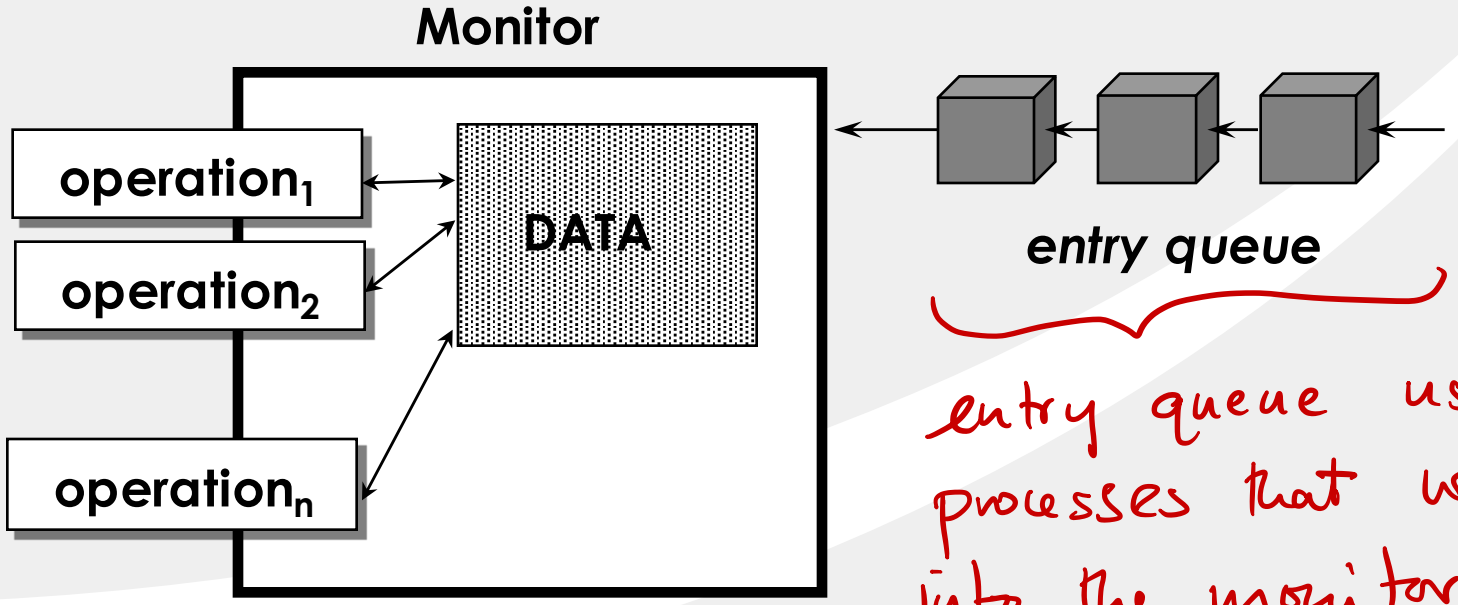
- A monitor is a high-level (programming language) abstraction that combines (and hides) the following:
  - shared data
  - operations on the data
  - synchronization using condition variables

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



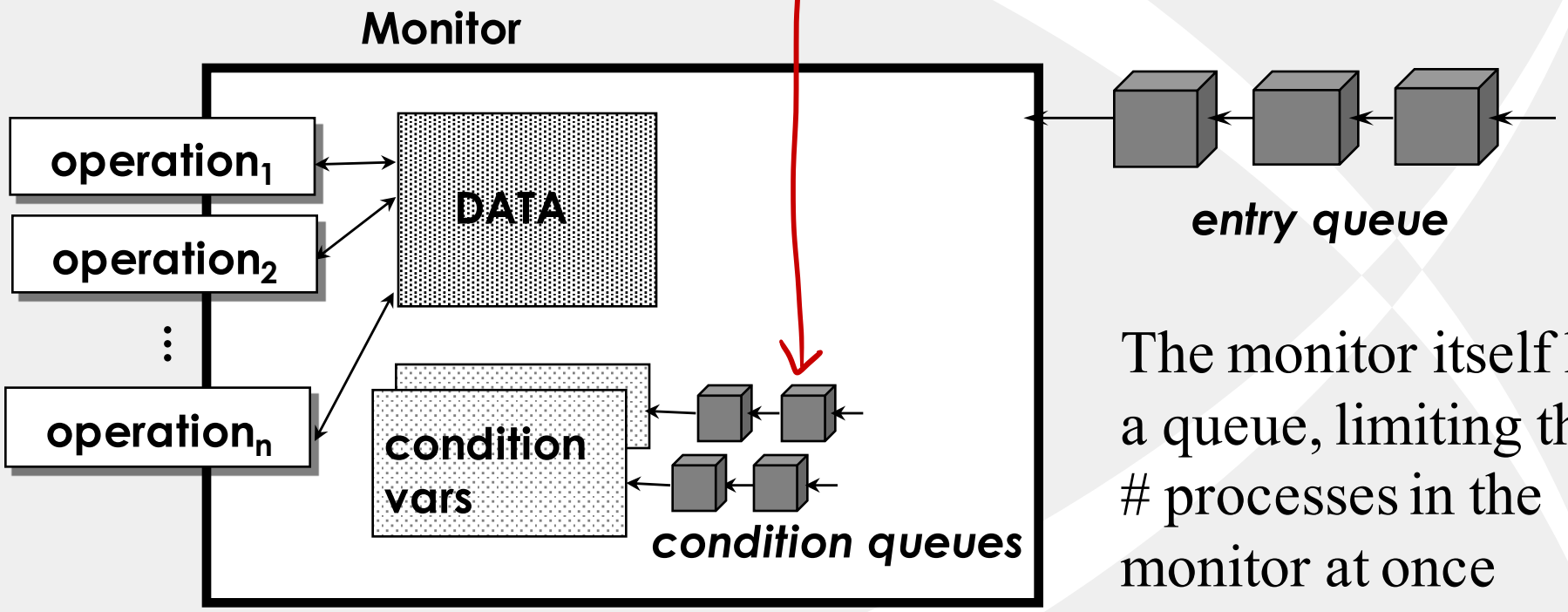
entry queue used by processes that want to get into the monitor but unable because the monitor is busy.

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

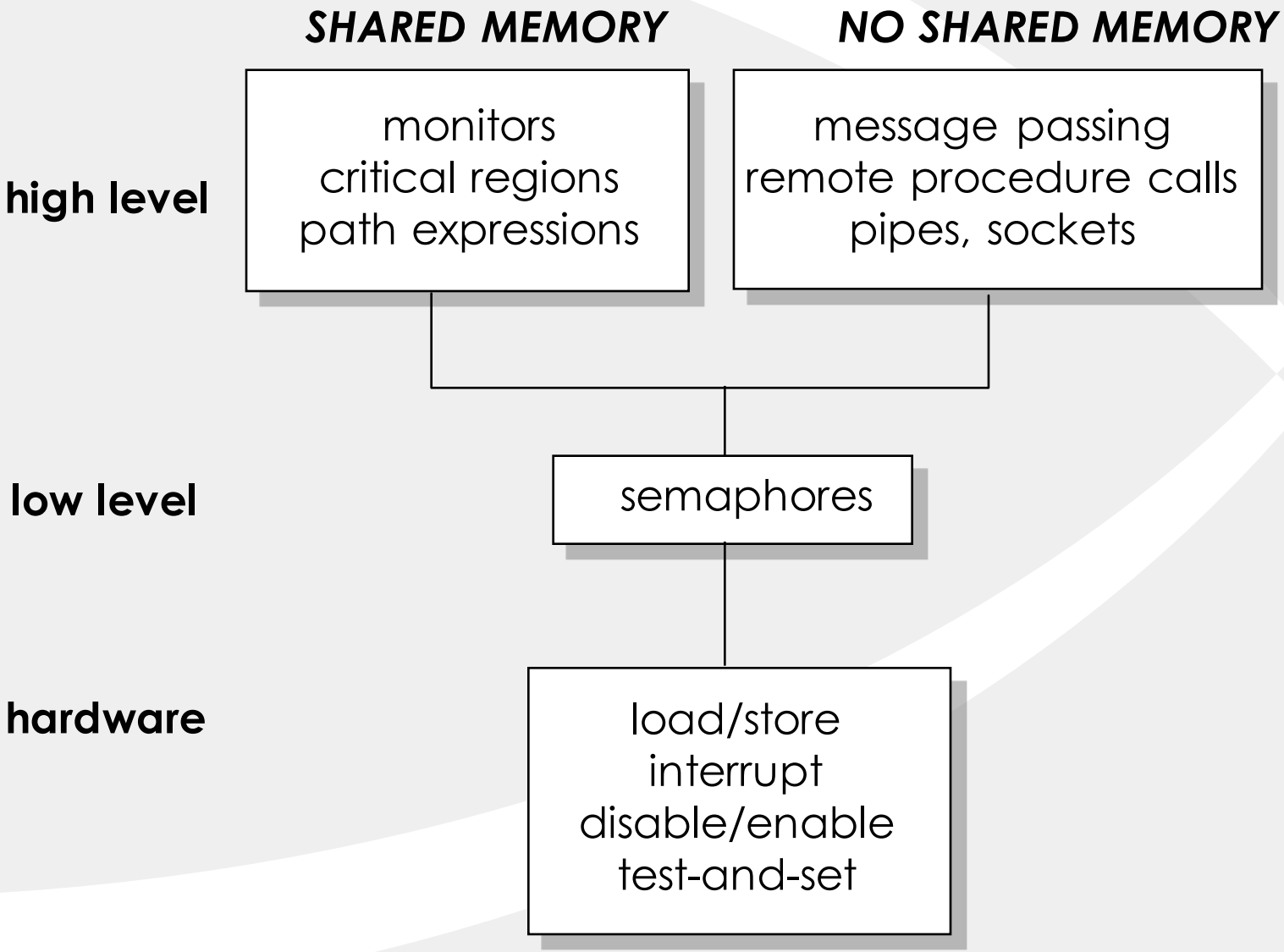
Problems with *synch* Primitives

- *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

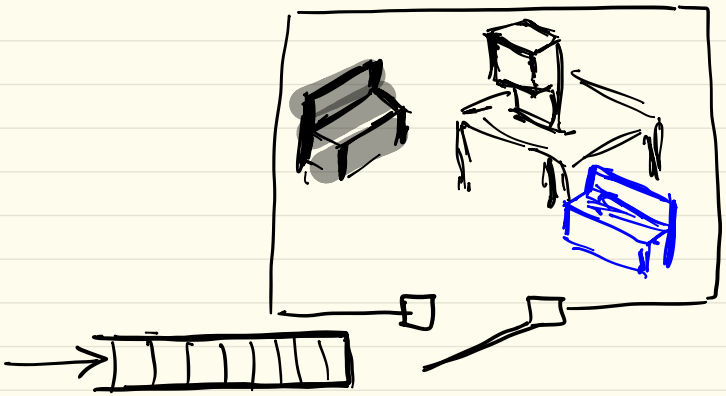
Problems with *synch* Primitives

- 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

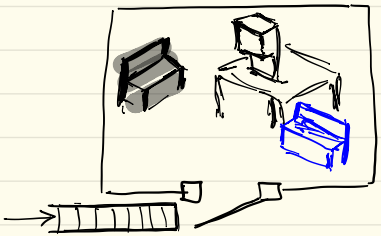
- enter the shop
- put ice creme
  * if drinkers on black bench wake them
- leave the shop.

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 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 *

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 one 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 wakeup that persn
else
let a new persn 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?

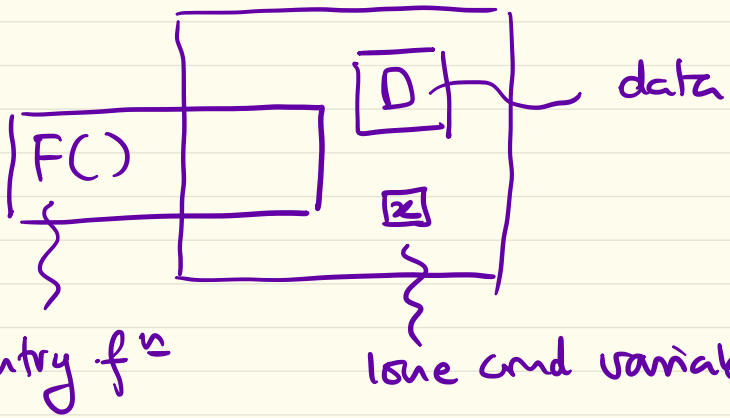
- First ice creme - always.
- Second ketchup - optionally - some drinkers want both some just ice creme

- What modifications?

After giving away ice creme for free for a long time, the ice creme 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's
- 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

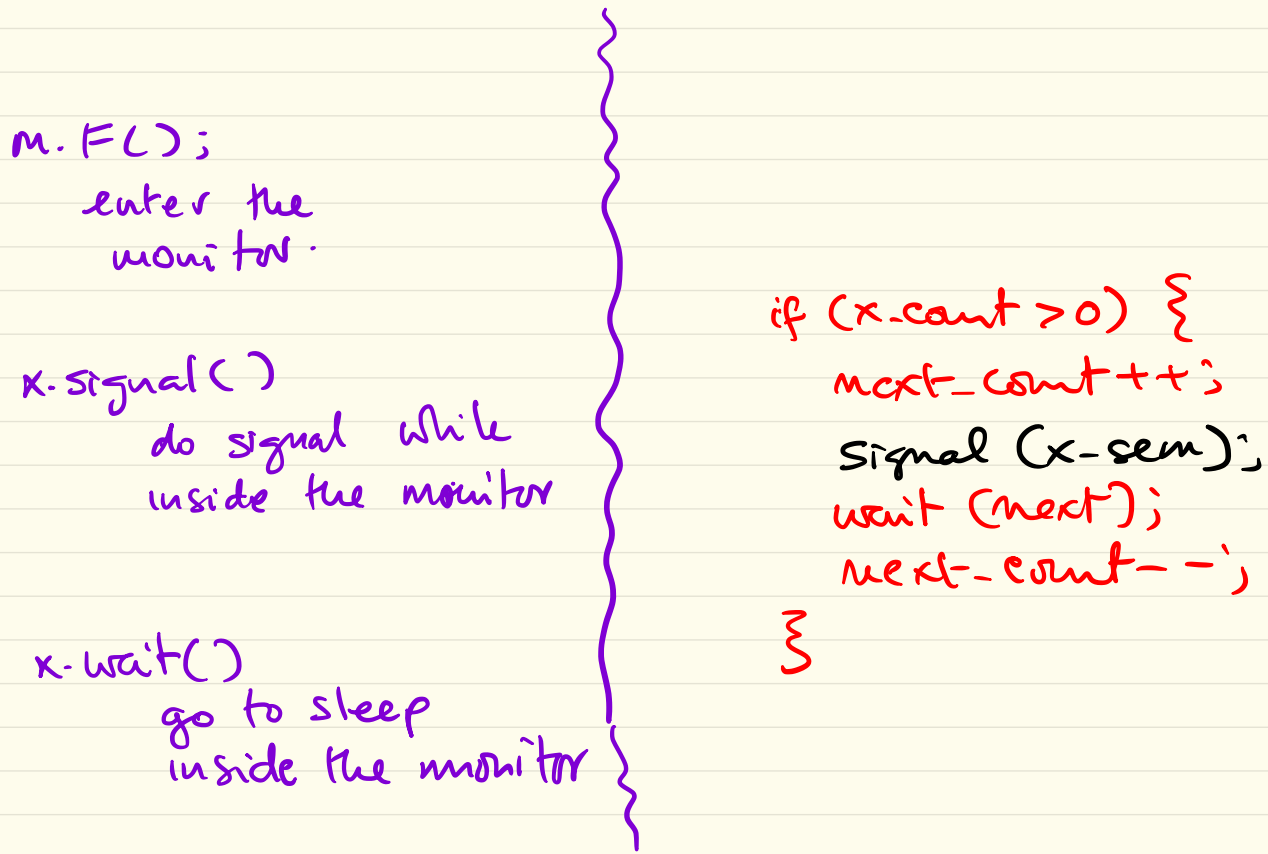
x.wait()
go to sleep
inside the monitor

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 (next_count > 0)
    signal (next)
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_count++;
if (next_count > 0)
    signal(next)
else
    signal(mutex);
wait(x_sem);
x_count--;
```

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.

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

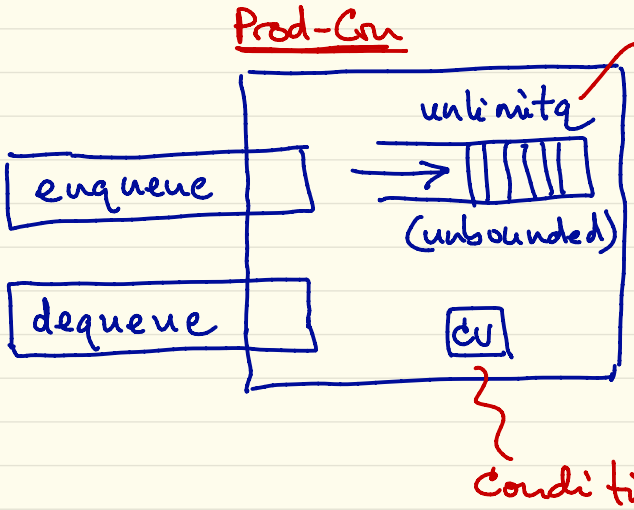
```
Monitor Semaphore {
    condvar semwait;
    int count;

    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 (1) {
        produce() → item
        pc.enqueue(item);
    }
}

Consumer () {
    while (1) {
        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.

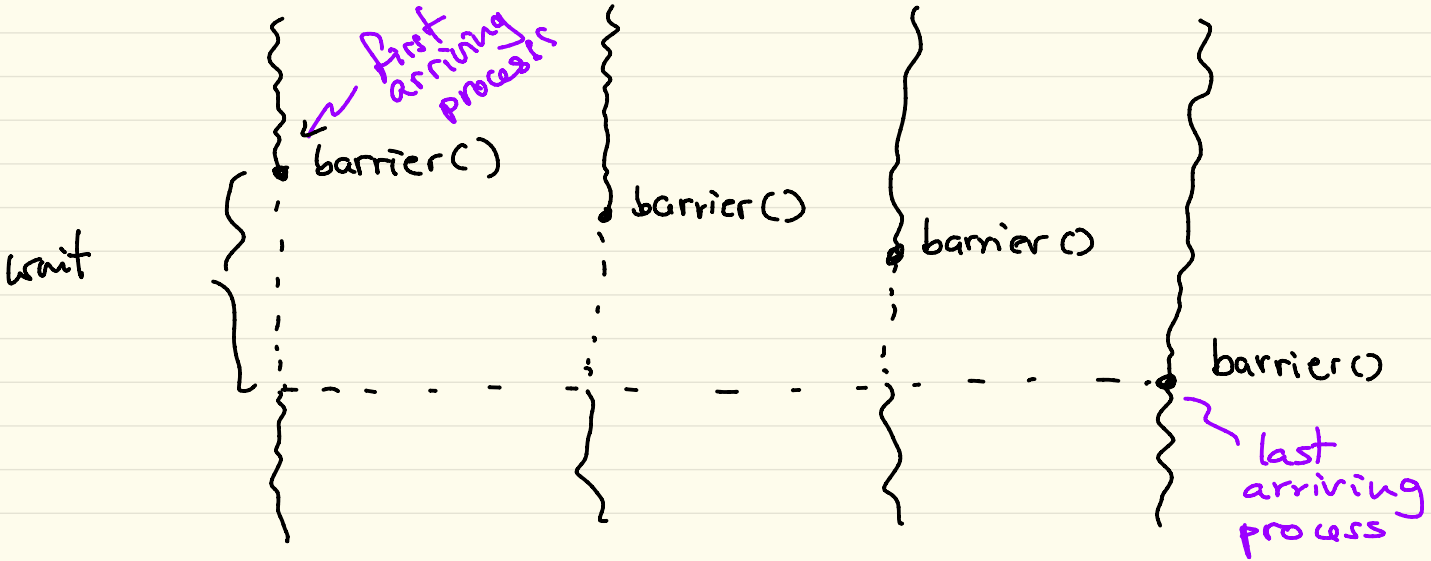
```
Monitor Prod-Con {
    condvar emptybuf;
    queue unlimitq;

void enqueue (item) {
    unlimitq.push (item);
    emptybuf.signal();
}

item dequeue () {
    if (unlimitq.size() == 0)
        emptybuf.wait();
    return unlimitq.pop();
}
```

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

A process trying 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 kth 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 "pri tized 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.

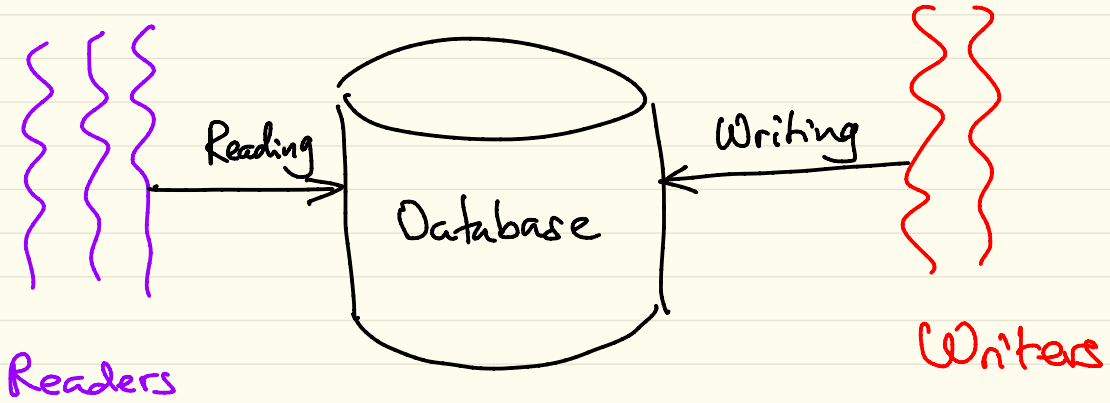
psignal(5) will release a waiting process with the highest process below Level 5. For example, if 3 processes are waiting at levels 3, 6, and 8.

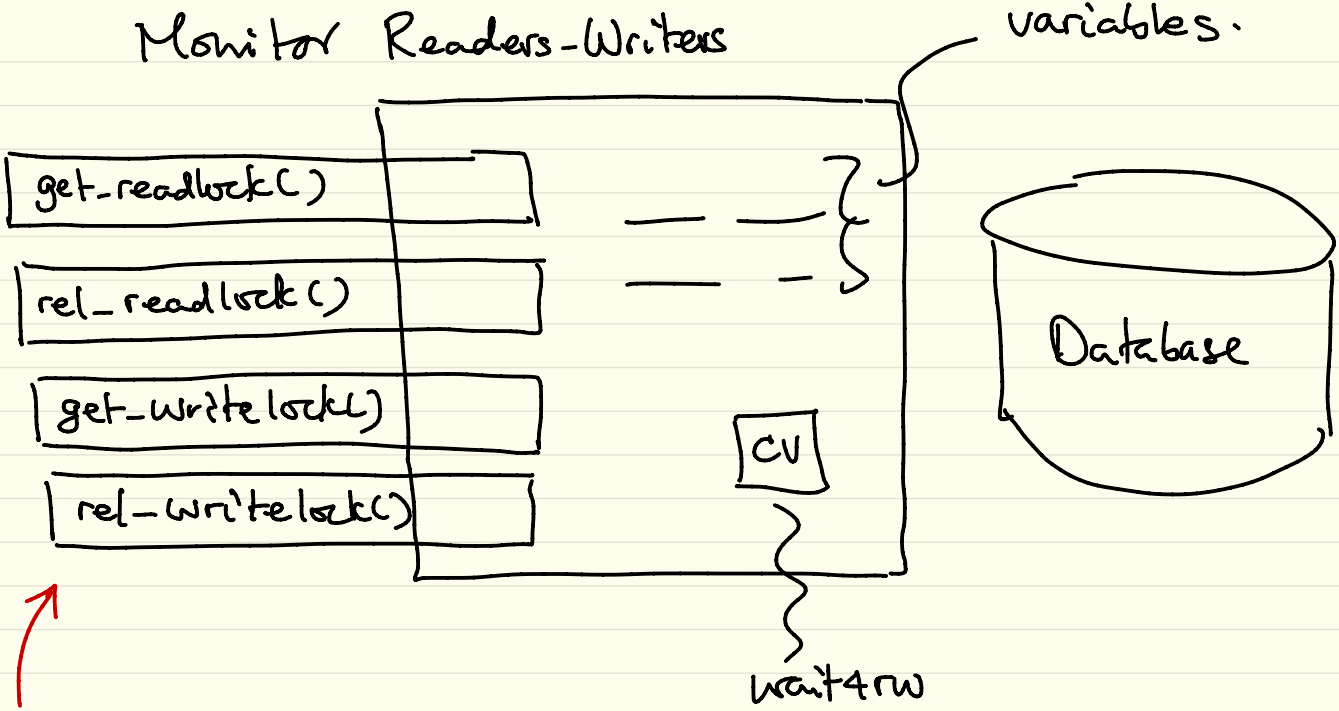
psignal (5) will release the process at level 6. So a signal at level 0 is bound to wake up a process if one is waiting. A signal at any other level can wake up no process because all of the waiting processes are at an higher level than the signalling level.

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.


Reading
Database
Writing
Readers
Writers



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

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar wait4rw;

    get_readlock() {

    }

    release_readlock() {

    }
```

```
    init() {
        readcnt = 0;
        reading = false;
        writing = false;
    }

    get_writelock() {

    }

    release_writelock() {

    }
```

R
- get_lock()
- Read DB
- rel_lock()

W
- get_wlock()
- Write
- rel_wlock()

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar wait4rw;

    get_readlock(){

    }
    release_readlock(){

    }

    get_writelock(){

    }
    release_writelock(){

    }
}
```

```
get_readlock() {
    readcnt++;
    if (reading)
        return;
    if (readcnt == 1
        & !writing ){
        return;
    }
    wait4rw.wait()
}
```

reading = true

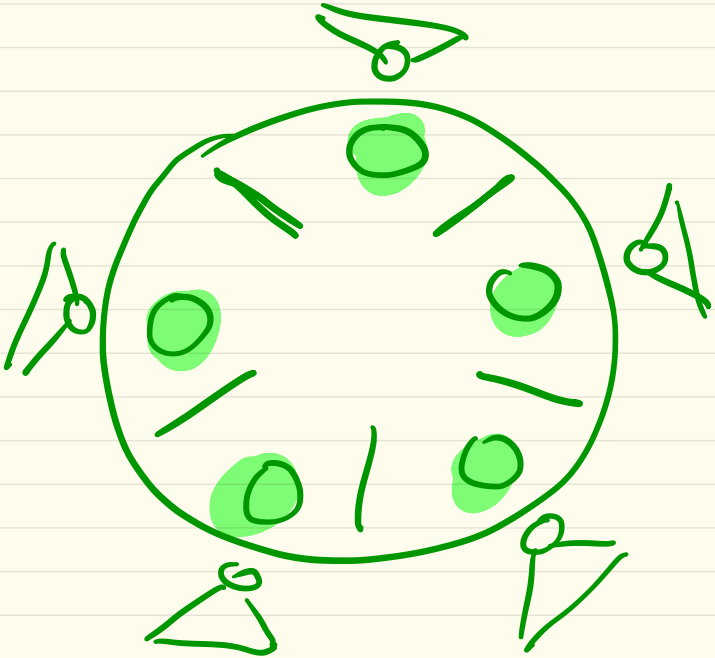
```
rel_readlock() {
    readcnt--;
    if (readcnt == 0){
        reading = false
        wait4rw.signal()
    }
}
```

```
Monitor Readers-Writers {
    int readcnt;
    bool writing, reading;
    condvar waitQrw;

    get_readlock(){
    }

    release_readlock(){
    }

    get_writelock(){
    }

    release_writelock(){
    }
```

```
get_writelock() {
    if (reading ||
        writing)
        waitQrw.lock();
    writing = true;
}
```

```
rel_writelock() {
    writing = false;
    waitQrw.signal();
}
```

Implement a monitor to solve the dining philosophers problem.

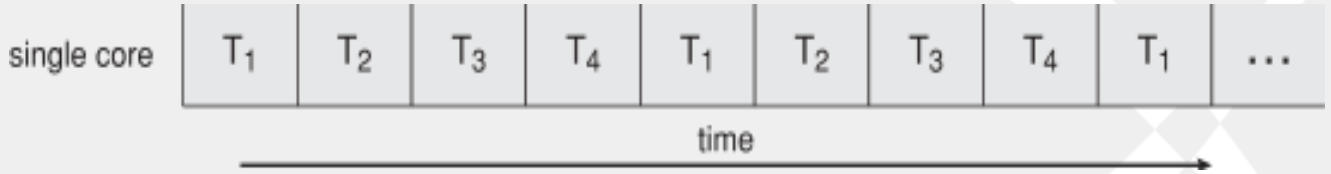


eat
think

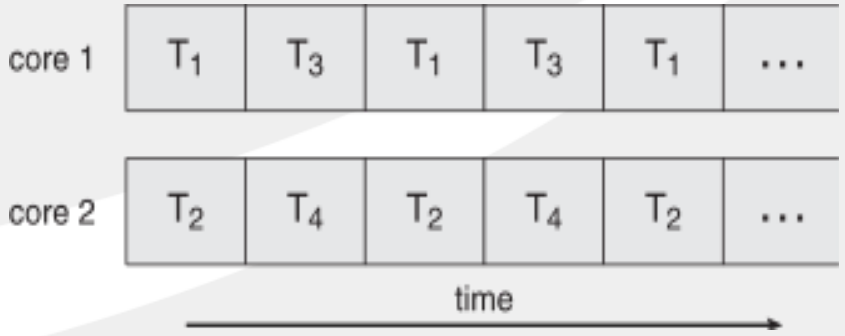
Processes and Threads (3)

Concurrency vs. Parallelism

- Concurrent execution on single-core system:

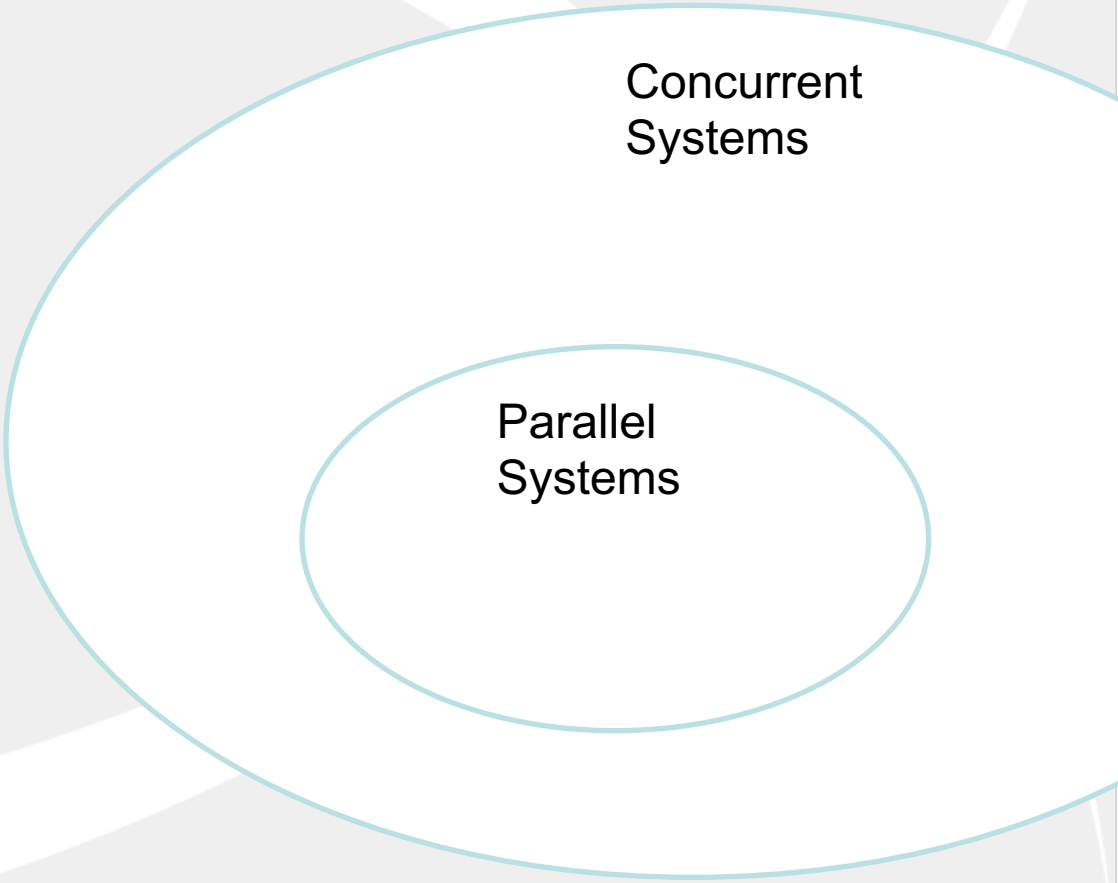


- Parallelism on a multi-core system:



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



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

- Application are not completely parallel
  - Serial portion
  - Parallel portion

Example

Amdhal's Law

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

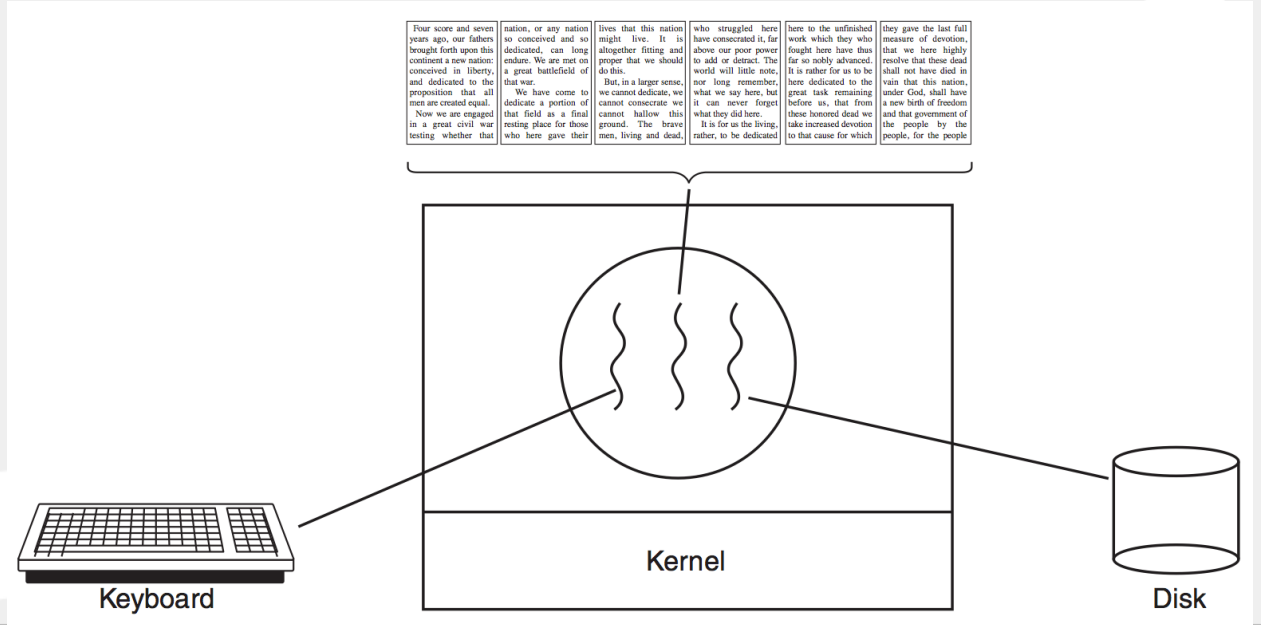
Speedup for an Application

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

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

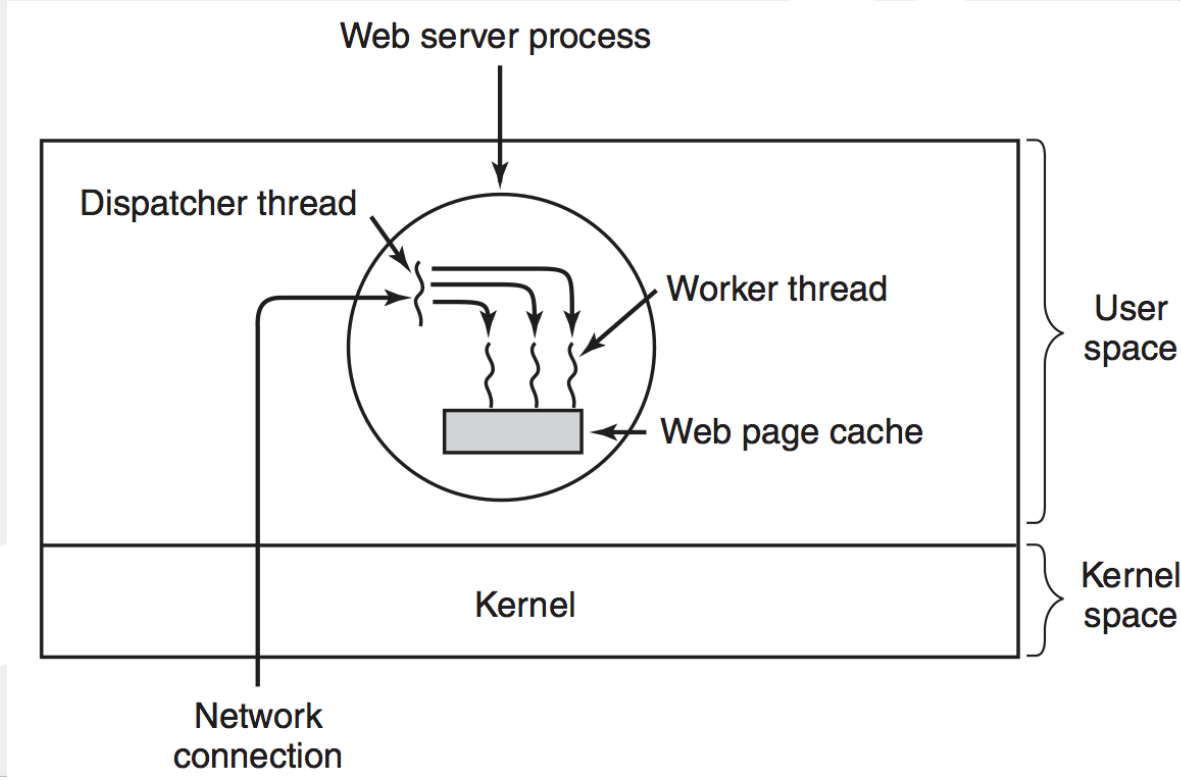
Motivation for Threads

- 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



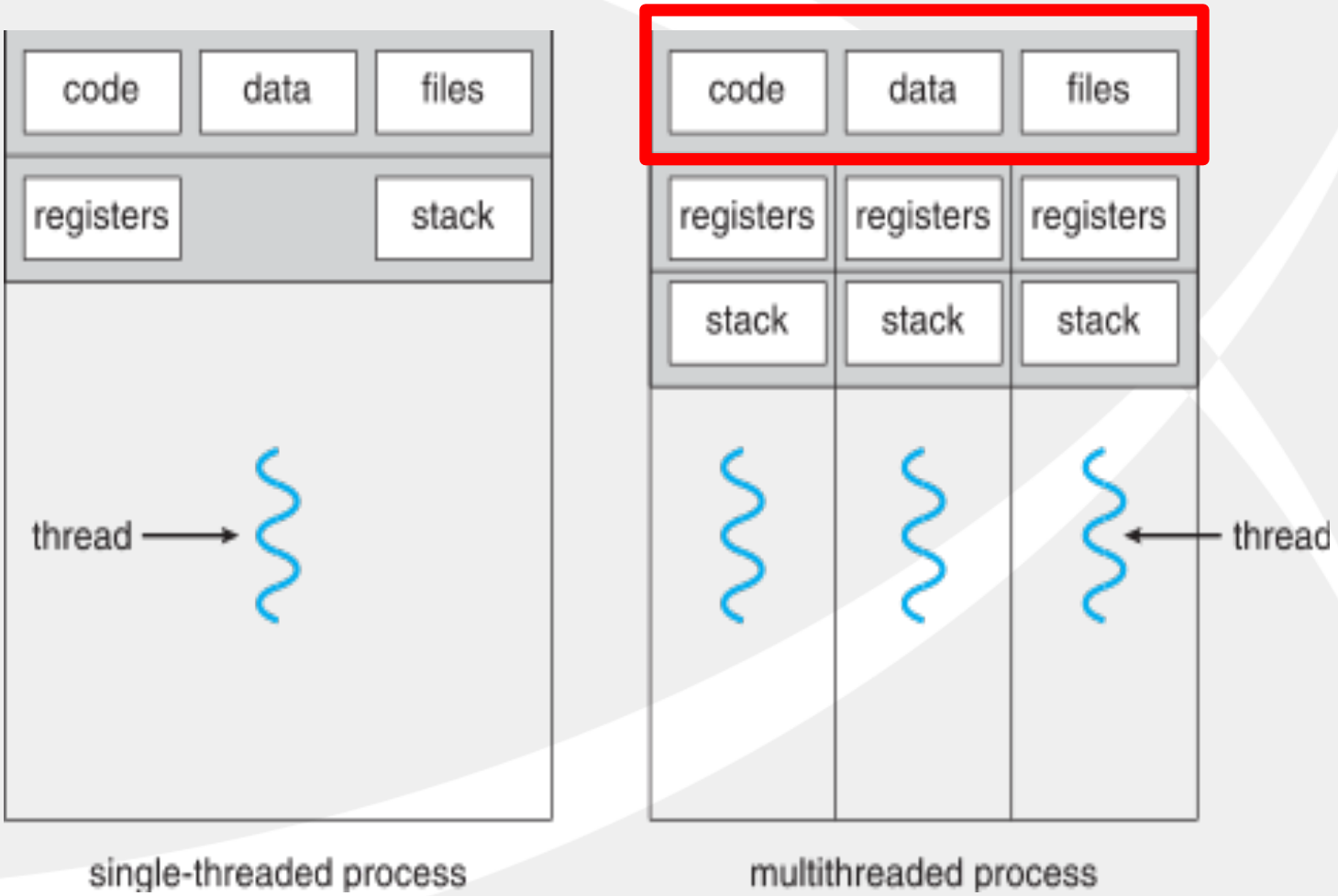
Why Not Use Processes?

- 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

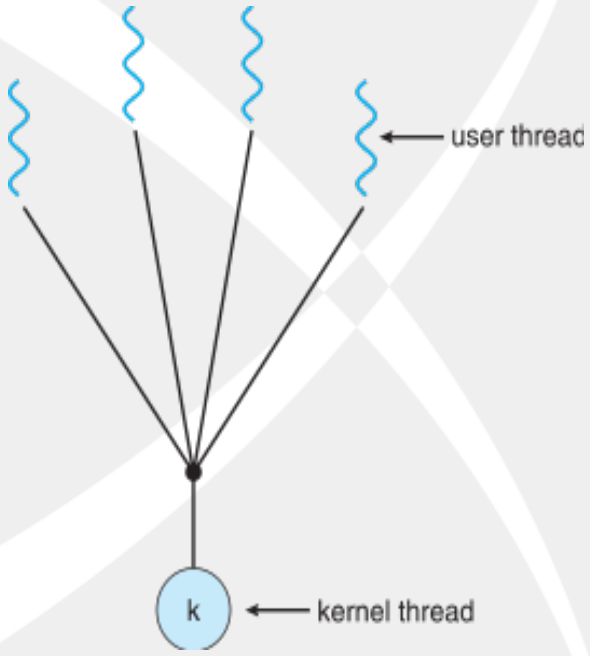


User vs. Kernel Threads

- **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

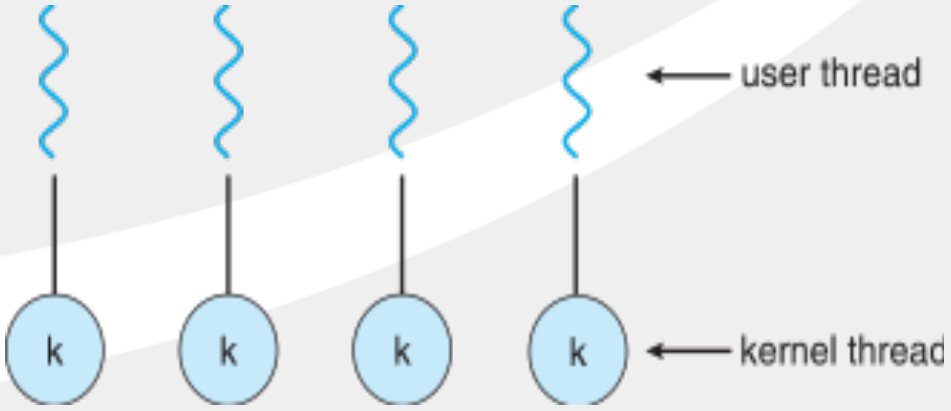
User Level Threads

- 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**



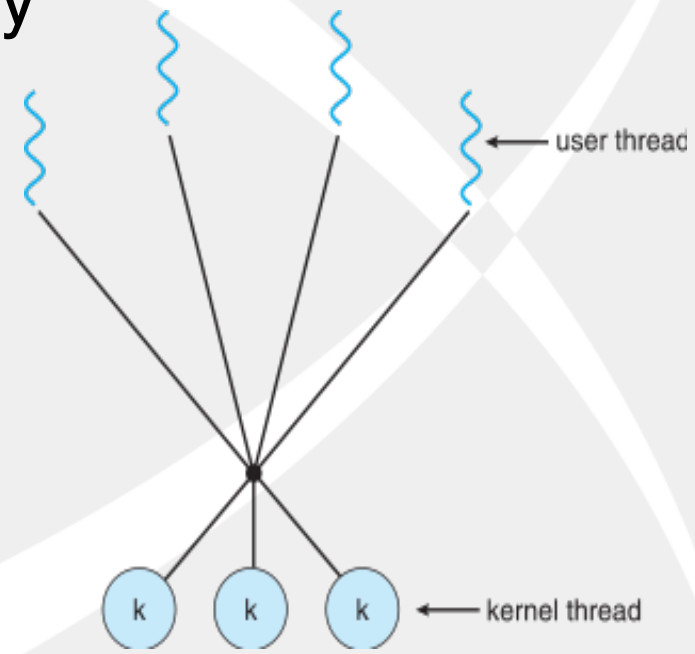
Kernel-Level Threads

- 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

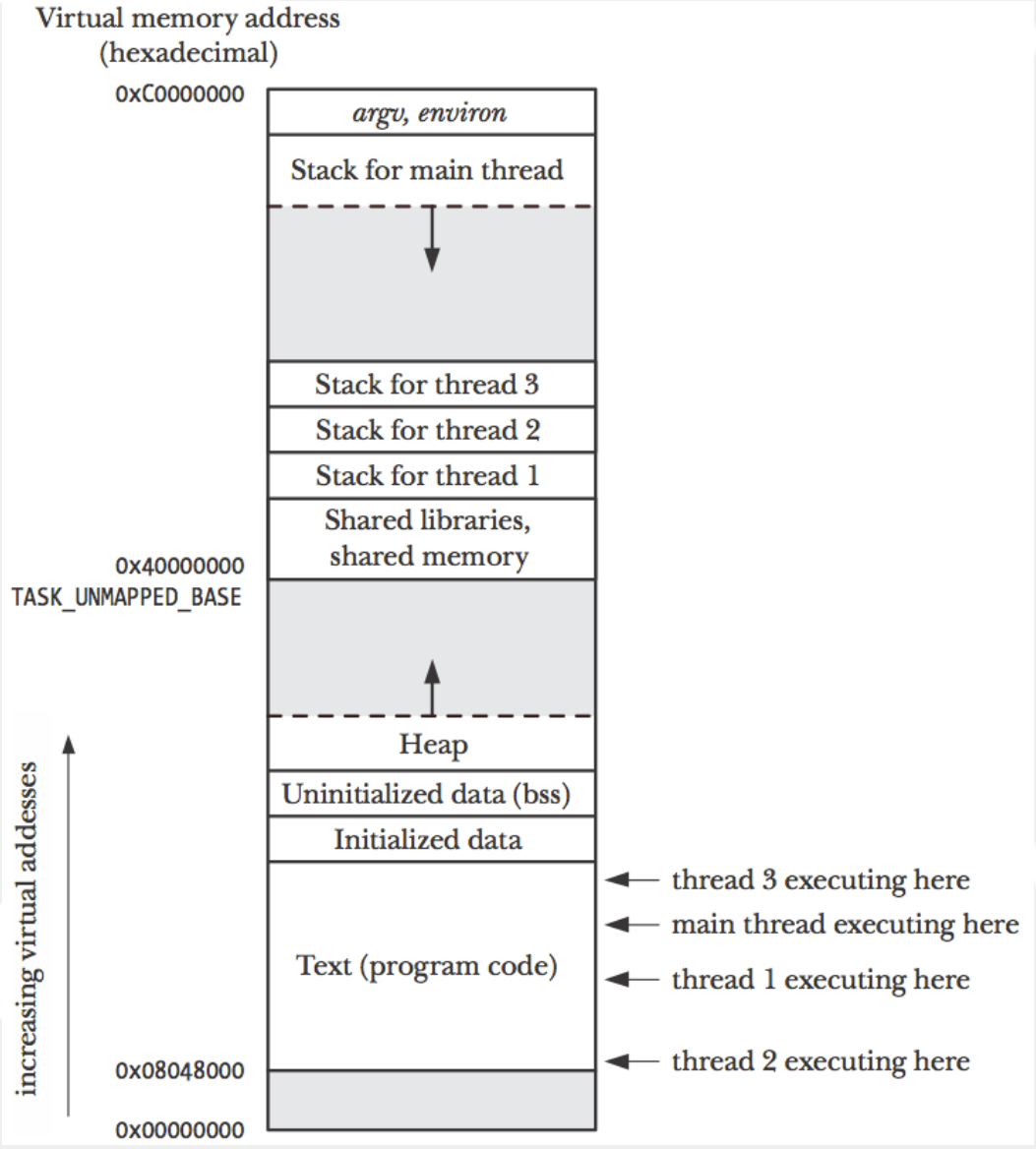


Thread Libraries

- **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 incre
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 t[illegible] cancellation remains pending until thread enables it

Thread Cancellation...

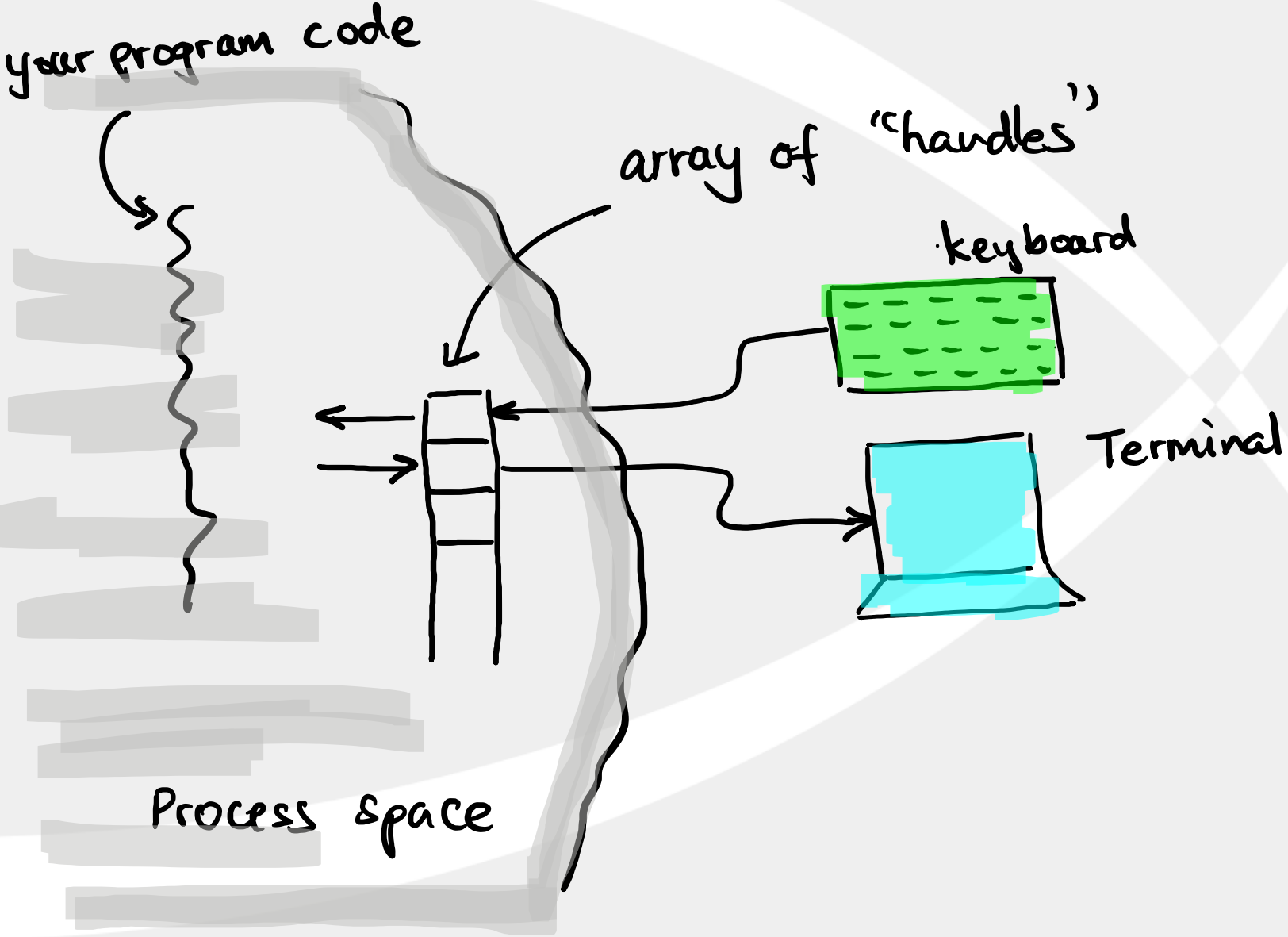
- Default type is deferred
  - Cancellation only occurs when thread reaches **cancellation point**
    - I.e. `pthread_testcancel()`
    - Then **cleanup handler** is invoked
- On Linux systems, thread cancellation is handled through signals

Processes and Threads (2)

Process I/O: How Does it Work?

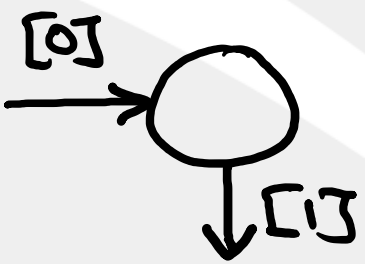
- Each process has an array of “handles”
- Each handle corresponds to an external device
- For example, you want to print something to screen, you write to handle #1
- You want to read something from keyboard, you read from handle #0
- You can setup handles to point to your data file and start reading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

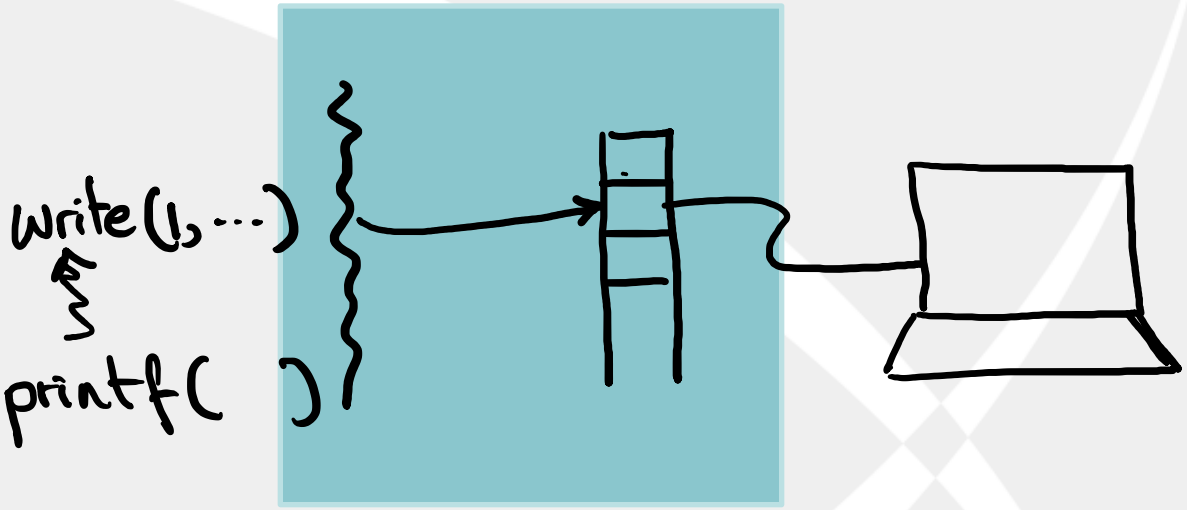
Process I/O: “Handles”

- 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

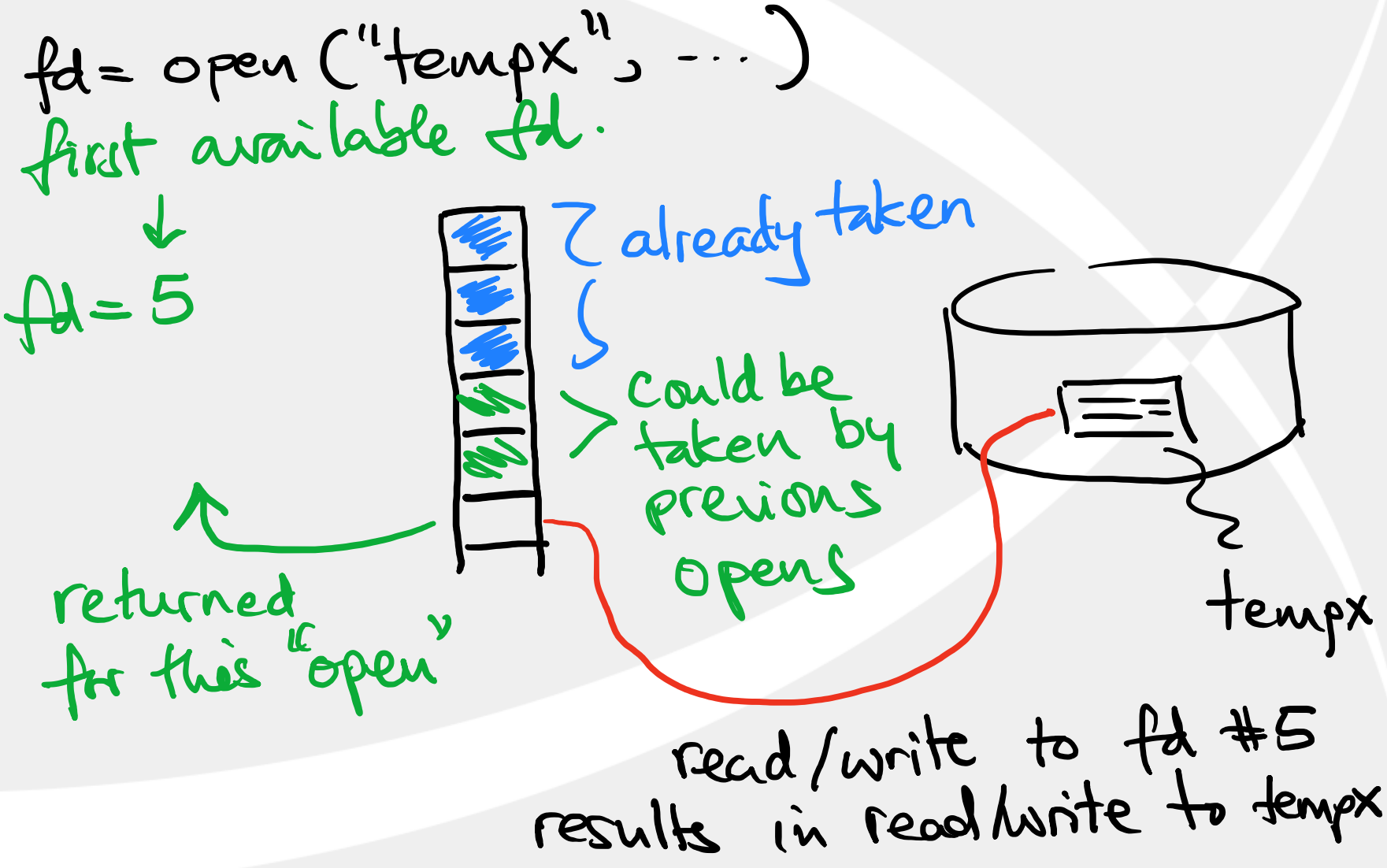
- File descriptor #0 (stdin): is the keyboard or whatever the standard input is
- File descriptor #1 (stdout): is the screen, printer or whatever the standard output is
- File descriptor #2 (stderr): is the error device which is mostly the same as stdout
- By having error separate from stdout, we can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



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

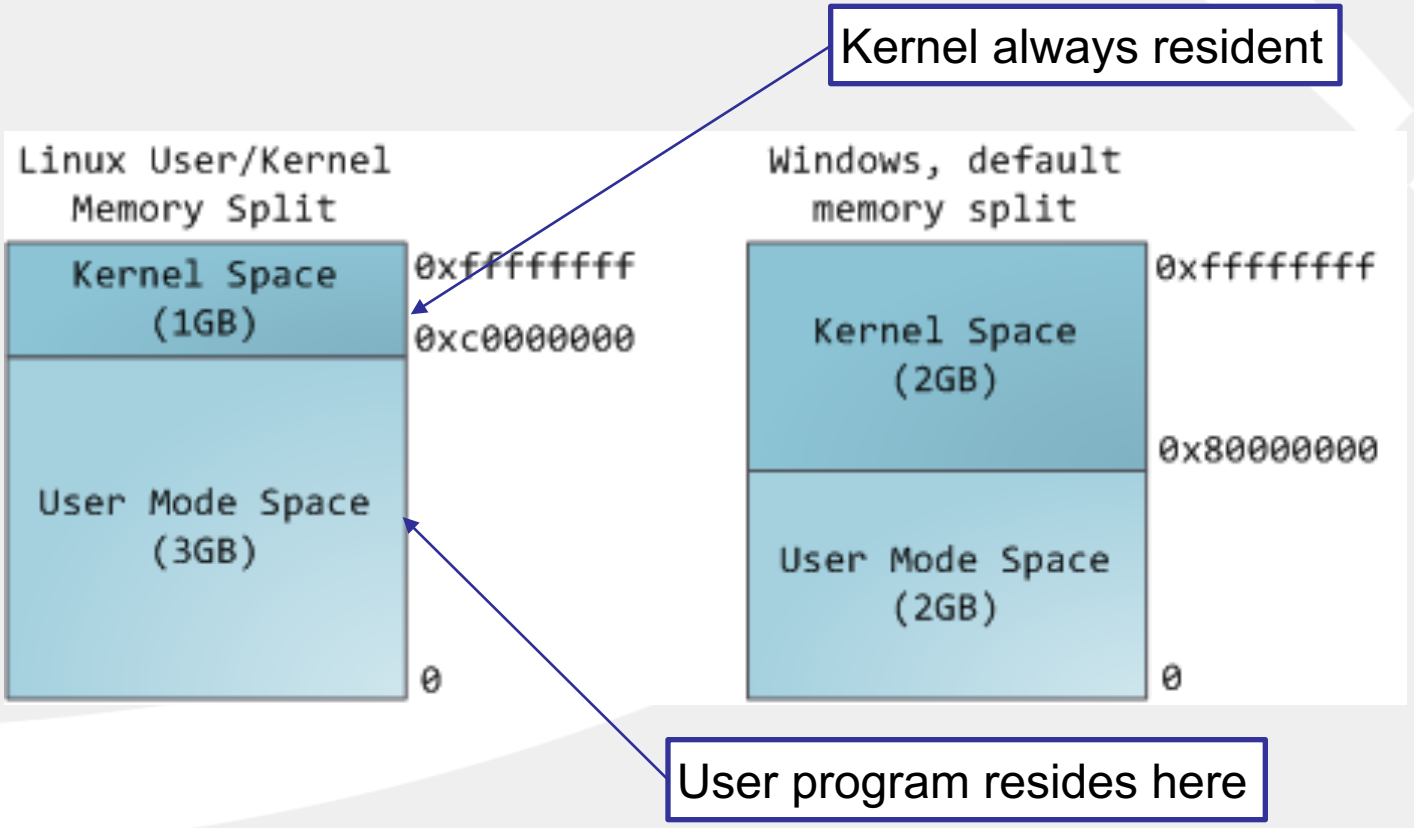
- We already discussed that a process has many elements
- One of the questions we encountered is where all the elements go
- Some of them go inside the processor
- Others go in the memory
- Obviously the memory needs to be organized in some way to put them

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

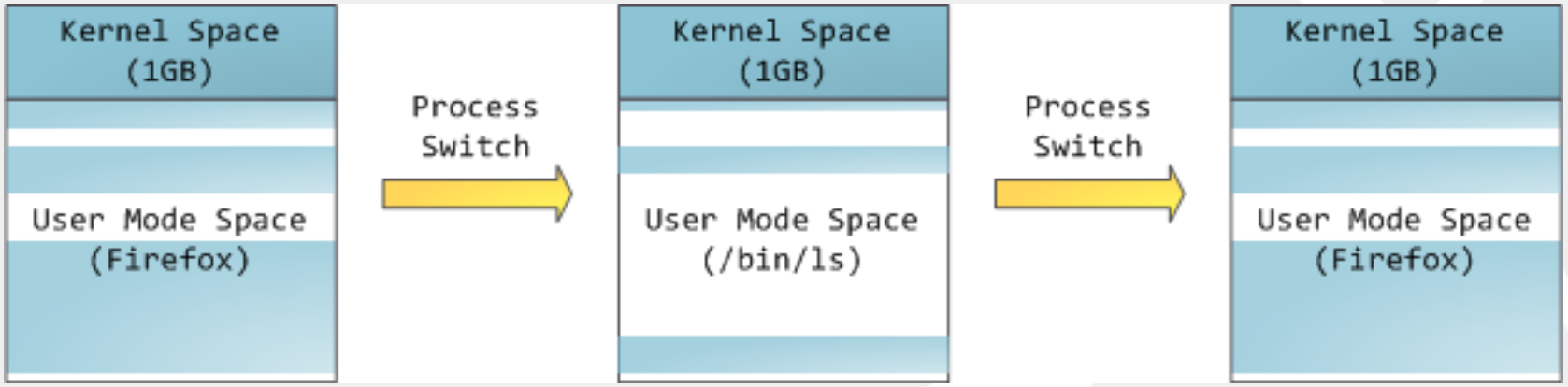


Address Space

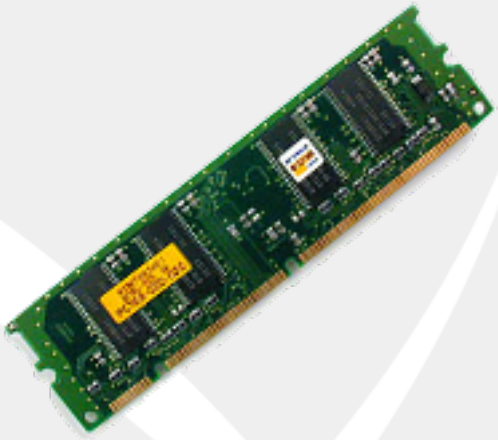
- When a process runs, all the addresses it could generate (full address space) belong to it – not shared with any other process
- Catch – part of it is taken by the kernel – the space mapped to the kernel – is persistent
- Using the kernel space – a process could communicate with other processes, how?

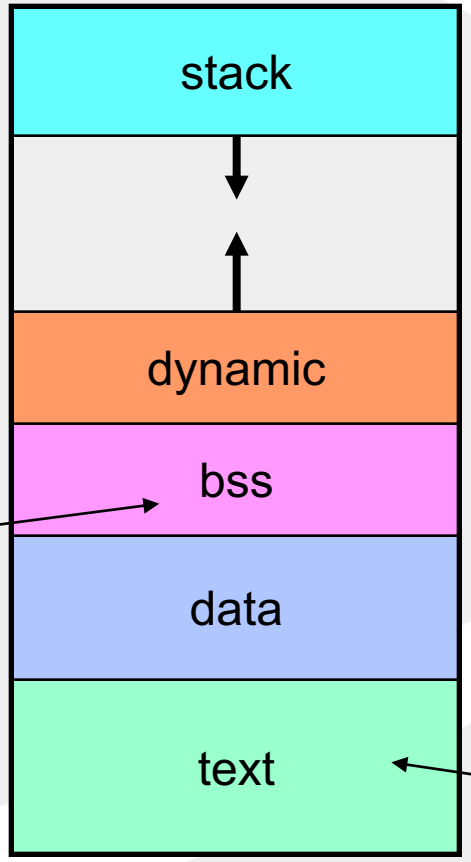
Address Space Switching

- Address space switching happens with a process switch



Address Space: Details of the User Space



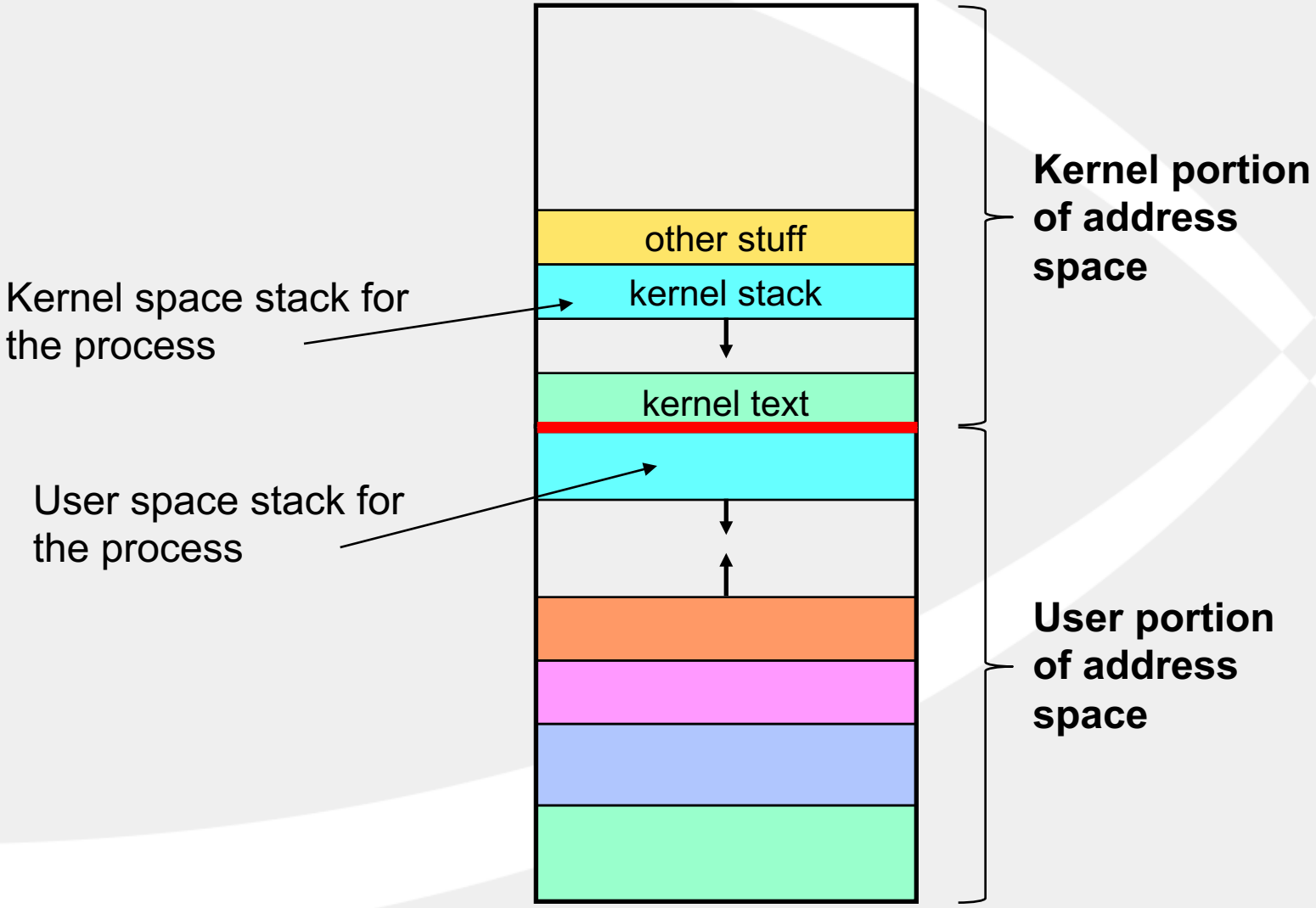


Block Started by Symbol:
Includes space for uninitialized variables

Read only segment that contains the program instructions

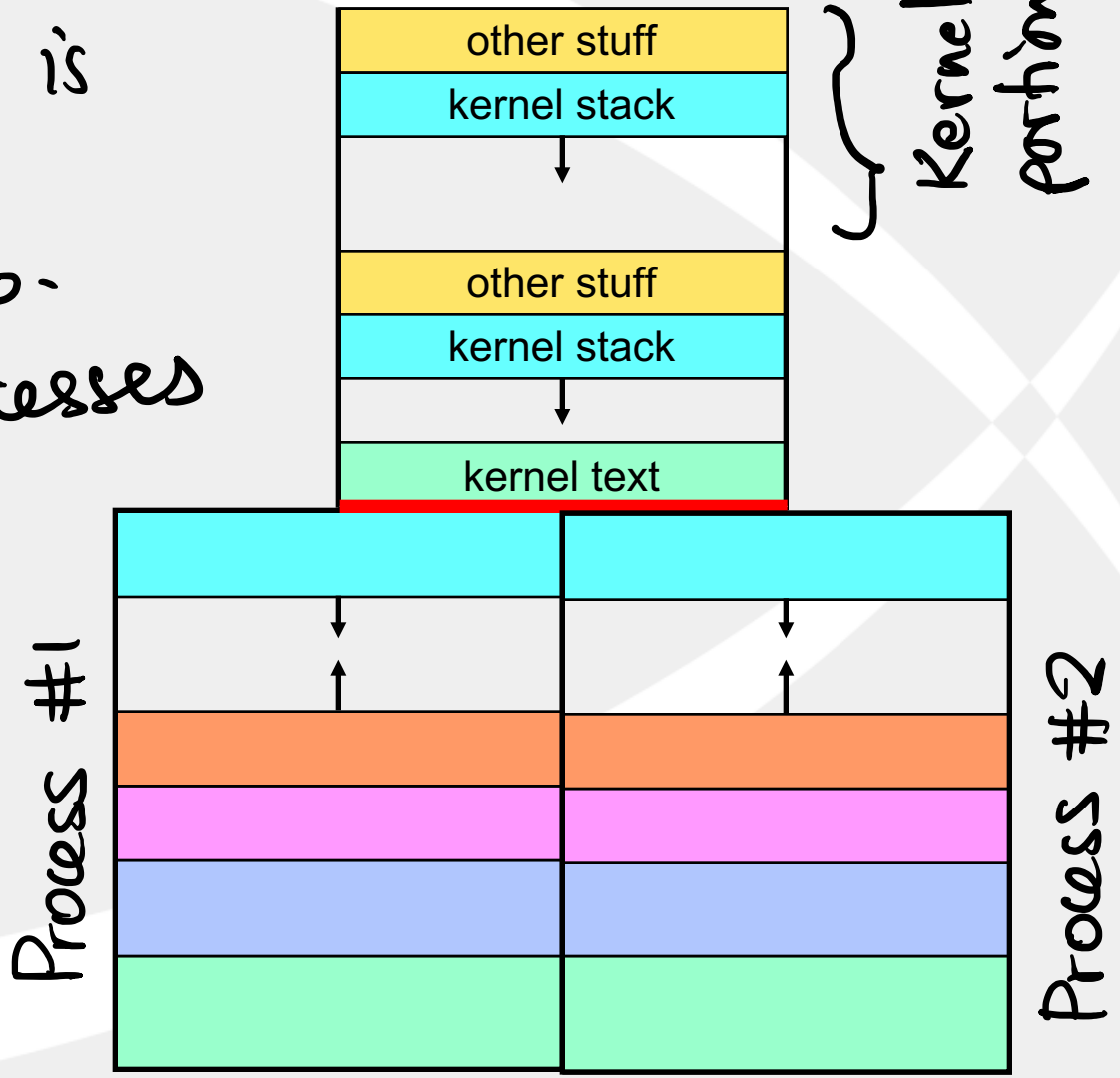
Address space occupied by user programs when **address space is NOT randomized**

Process Address Space



Process After Fork()

Kernel space is suitable for sharing info. between 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

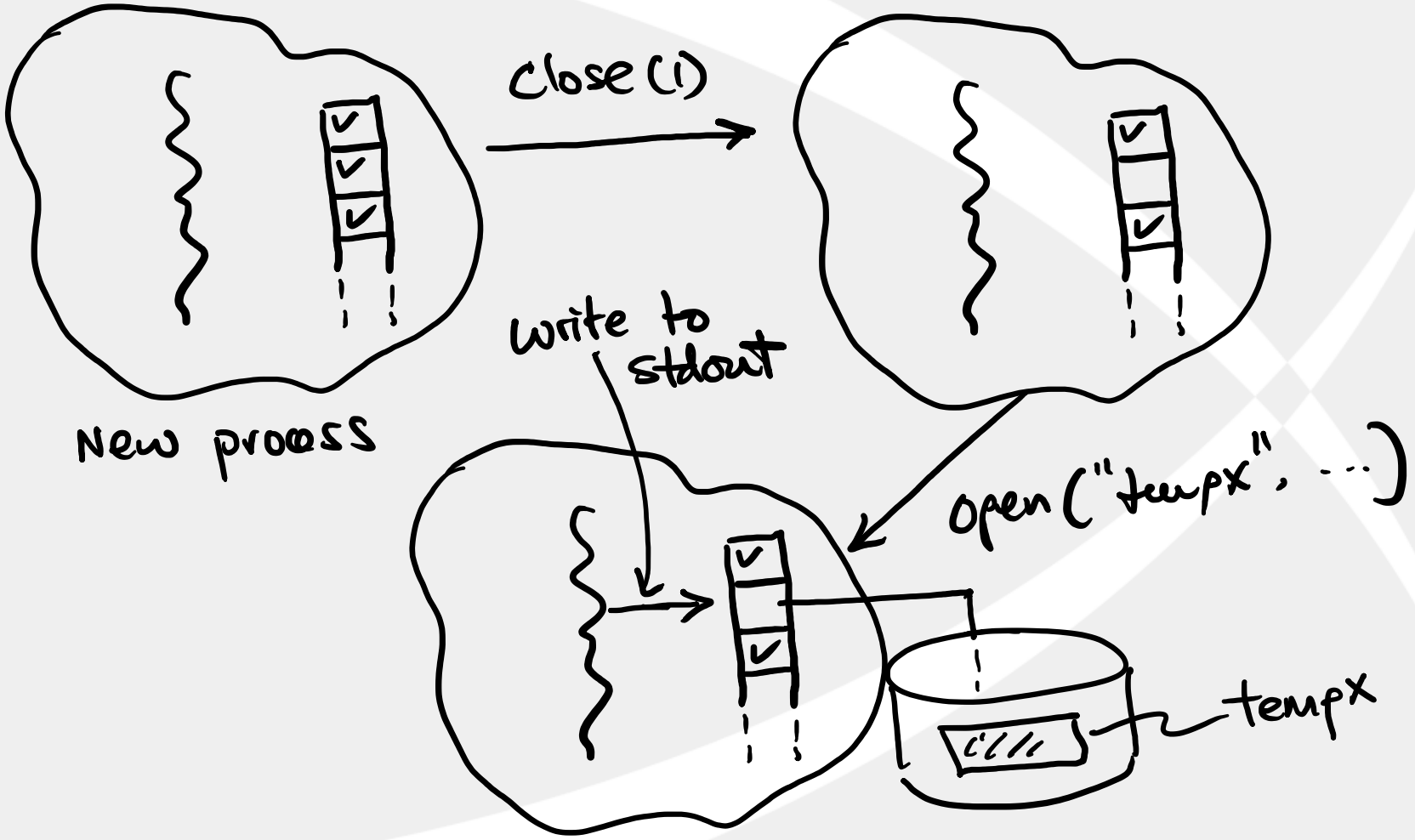
Important Note on Application Launching

- 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

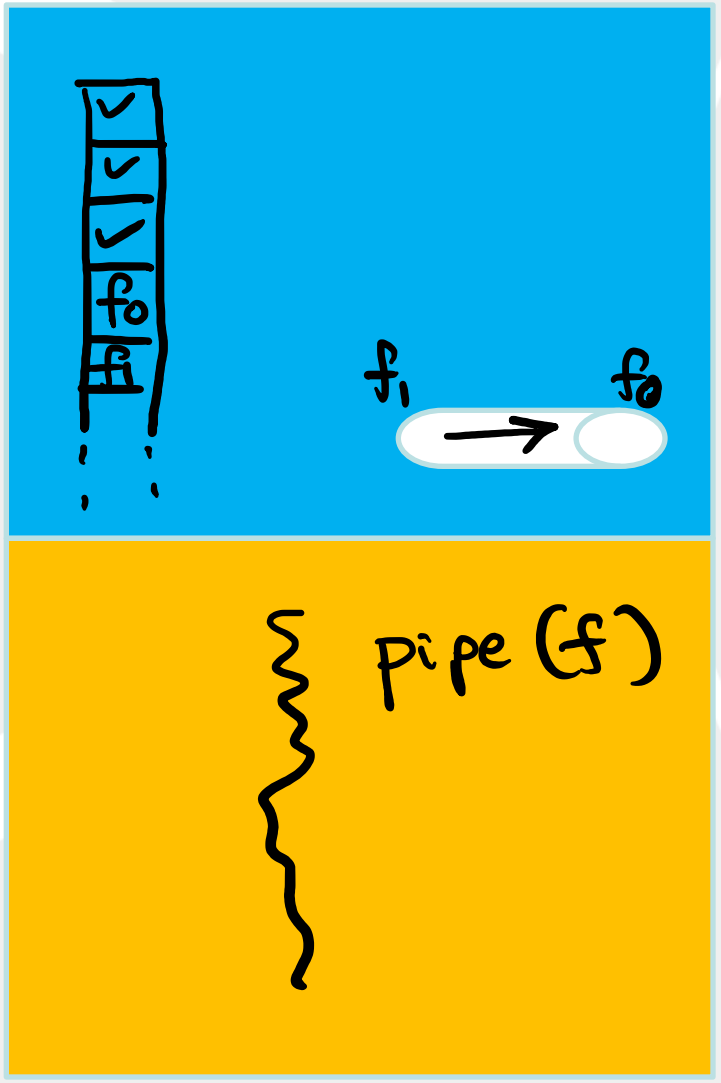
- Implement input redirection

Command Piping

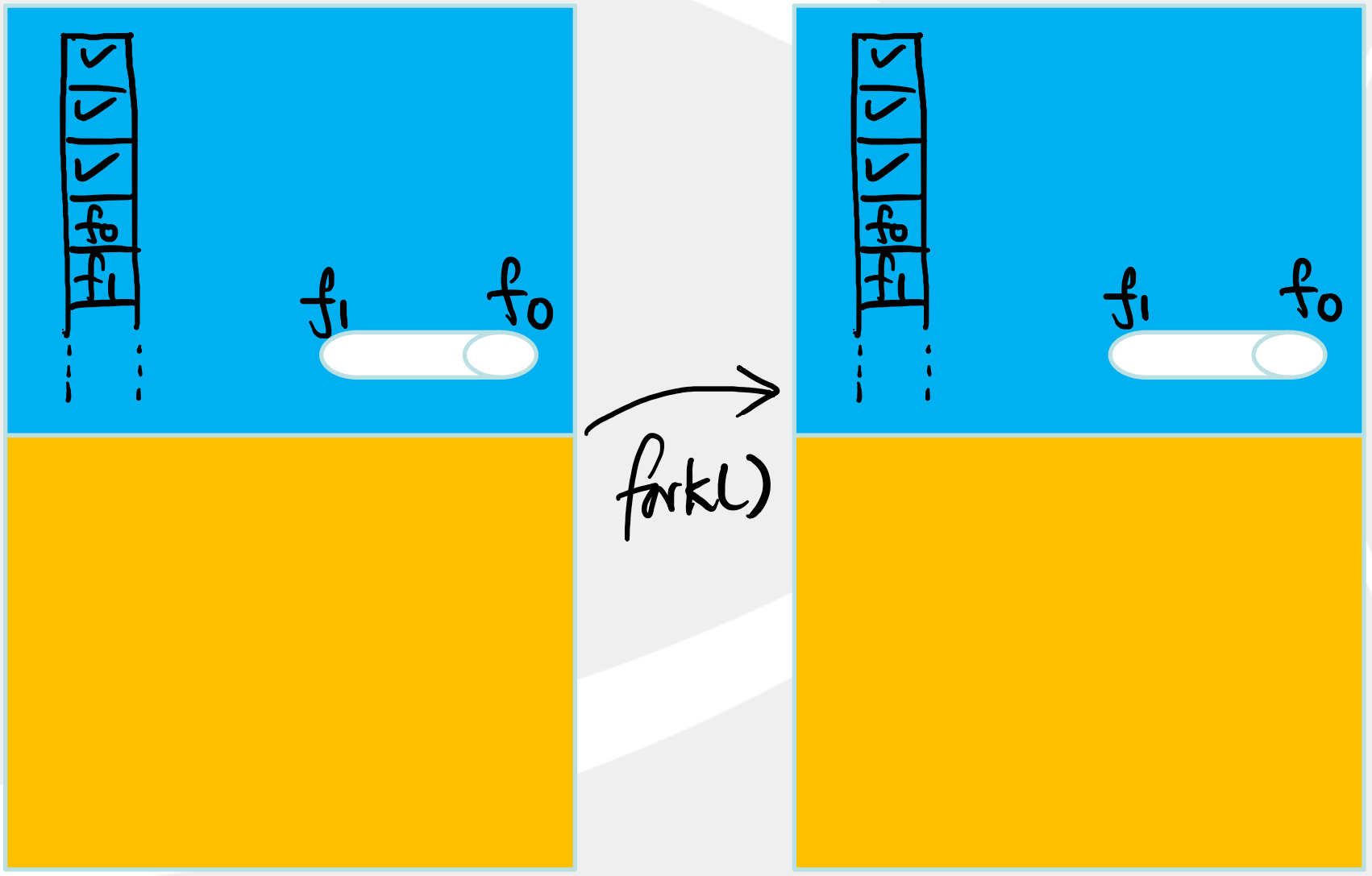
- Goal:
  - Connect the output of a process to the input of another process
- Problem:
  - Processes are independent of each other
- Idea:
  - Do the transfer through the kernel space

Step 1: Create a Pipe

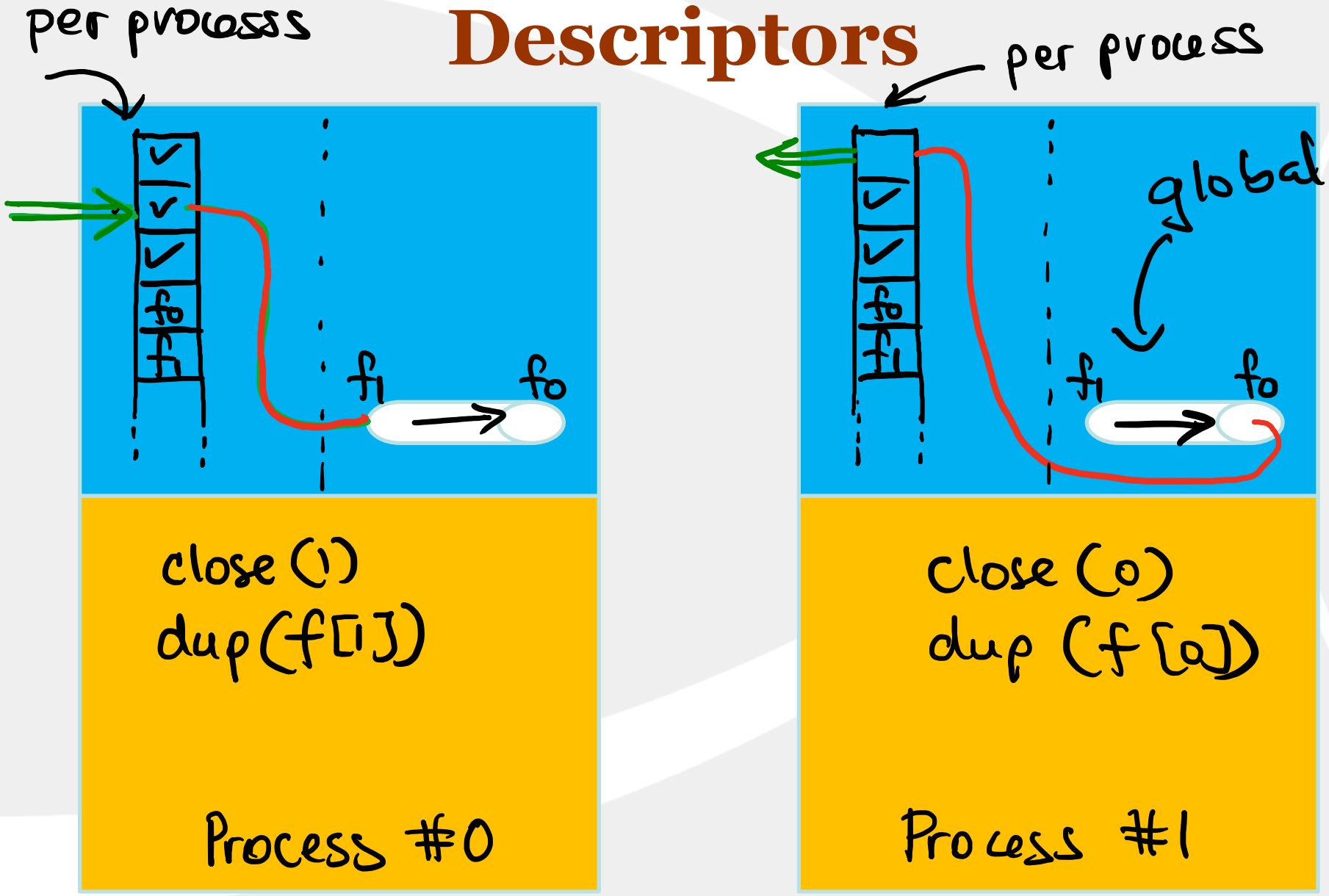
- Use the pipe() system call and create a pipe
  - Has two end points
  - Allows unidirectional data flow
  - Write end and read end



Step 2: Fork the Process



Step 3: Rewire the File Descriptors



Practice Questions #2

Page 81: 1, 3, 9, 10, 12, 13, 17, 18, 20, 21, 22, 28

Page 174: 2, 4, 5, 8, 11, 12

Practice Questions #2

Answers

Page 81 of Textbook

1

An operating system must provide the users with an extended machine, and it must manage the I/O devices and other system resources. To some extent, these are different functions.

3. In a timesharing system, multiple users can access and perform computations on a computing system simultaneously using their own terminals. Multiprogramming systems allow a user to run multiple programs simultaneously. All timesharing systems are multiprogramming systems but not all multiprogramming systems are timesharing systems since a multiprogramming system may run on a PC with only one user.

9

Consider fairness and real time. Fairness requires that each process be allocated its resources in a fair way, with no process getting more than its fair share. On the other hand, real time requires that resources be allocated based on the times when different processes must complete their execution. A real-time process may get a disproportionate share of the resources.

10. Most modern CPUs provide two modes of execution: kernel mode and user mode. The CPU can execute every instruction in its instruction set and use every feature of the hardware when executing in kernel mode. However, it can execute only a subset of instructions and use only subset of features when executing in the user mode. Having two modes allows designers to run user programs in user mode and thus deny them access to critical instructions.

12

Choices (a), (c), and (d) should be restricted to kernel mode.

13. It may take 20, 25 or 30 msec to complete the execution of these programs depending on how the operating system schedules them. If *P0* and *P1* are scheduled on the same CPU and *P2* is scheduled on the other CPU, it will take 20 msec. If *P0* and *P2* are scheduled on the same CPU and *P1* is scheduled on the other CPU, it will take 25 msec. If *P1* and *P2* are scheduled on the same CPU and *P0* is scheduled on the other CPU, it will take 30 msec. If all three are on the same CPU, it will take 35 msec.

17

A trap instruction switches the execution mode of a CPU from the user mode to the kernel mode. This instruction allows a user program to invoke functions in the operating system kernel.

18

. The process table is needed to store the state of a process that is currently suspended, either ready or blocked. Modern personal computer systems have dozens of processes running even when the user is doing nothing and no programs are open. They are checking for updates, loading email, and many other things, On a UNIX system, use the *ps -a* command to see them. On a Windows system, use the task manager.

20

. Fork can fail if there are no free slots left in the process table (and possibly if there is no memory or swap space left). Exec can fail if the file name given does not exist or is not a valid executable file. Unlink can fail if the file to be unlinked does not exist or the calling process does not have the authority to unlink it.

21

Time multiplexing: CPU, network card, printer, keyboard.
Space multiplexing: memory, disk.
Both: display.

Both: display.

22

If the call fails, for example because *fd* is incorrect, it can return −1. It can also fail because the disk is full and it is not possible to write the number of bytes requested. On a correct termination, it always returns *nbytes*.

28

. As far as program logic is concerned, it does not matter whether a call to a library procedure results in a system call. But if performance is an issue, if a task can be accomplished without a system call the program will run faster. Every system call involves overhead time in switching from the user context to the kernel context. Furthermore, on a multiuser system the operating system may schedule another process to run when a system call completes, further slowing the progress in real time of a calling process.

Page 174 of Textbook.

(2) You could have a register containing a pointer to the current process-table entry. When I/O completed, the CPU would store the current machine state in the current process-table entry. Then it would go to the interrupt vector for the interrupting device and fetch a pointer to another process-table entry (the service procedure). This process would then be started up.

(4) There are several reasons for using a separate stack for the kernel. Two of them are as follows. First, you do not want the operating system to crash because a poorly written user program does not allow for enough stack space. Second, if the kernel leaves stack data in a user program's memory space upon return from a system call, a malicious user might be able to use this data to find out information about other processes.

(5) The chance that all five processes are idle is 1/32, so the CPU idle time is 1/32.

(8) The probability that all processes are waiting for I/O is 0.4^6 which is 0.004096. Therefore, CPU utilization = $1 - 0.004096 = 0.995904$.

(11) No. If a single-threaded process is blocked on the keyboard, it cannot fork.

(12) A worker thread will block when it has to read a Web page from the disk. If user-level threads are being used, this action will block the entire process, destroying the value of multithreading. Thus it is essential that kernel threads are used to permit some threads to block without affecting the others.

Practice Questions #1

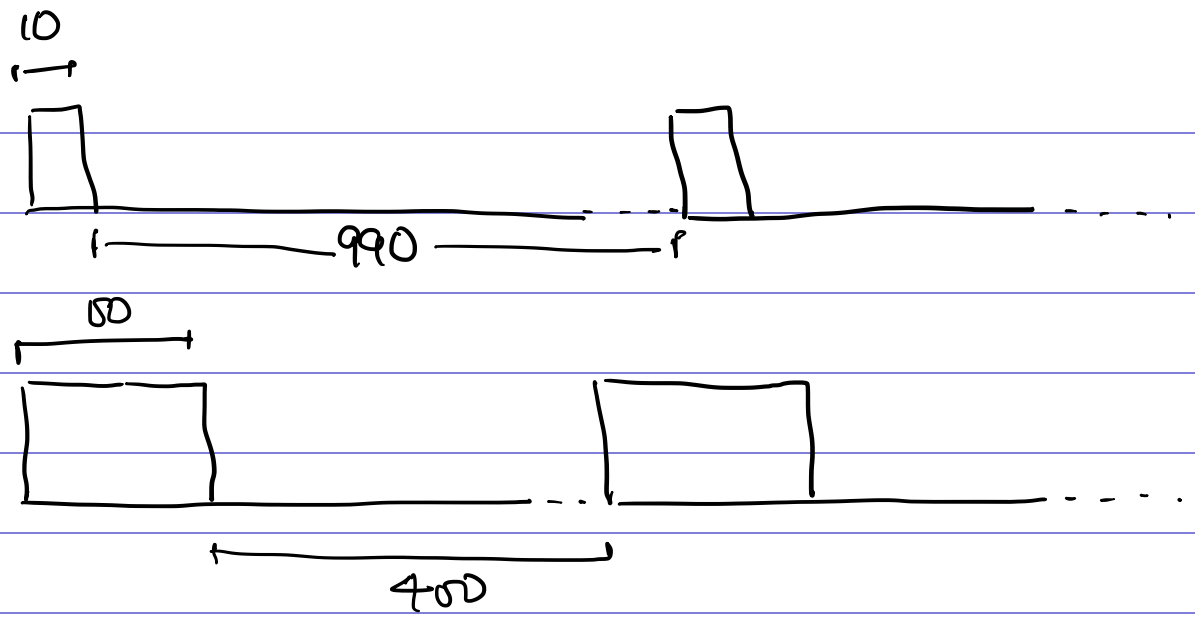
Answers.

(1) System call involves a mode switch from user to kernel. System call also runs procedures provided by the OS developer and not arbitrary functions written by the programmer.

(2) System calls involve the user-to-kernel switch not relevant for interrupts. Interrupt processing takes place even in microcontrollers that do not have an operating system. At interrupt, the interrupt service routine pointed to by the interrupt service vector is executed. In a modern OS, the interrupt service routines and vectors are managed by the kernel.

(3) Mode switching is an essential part of the system call. The purpose of the system call is to run procedures that need higher privileges. So a mode switch is necessary.

(4)



Utilization values we could get

$\frac{10}{500}$, $\frac{0}{500}$, $\frac{100}{500}$

(5) $\frac{110}{500}$, $\frac{10}{500}$, $\frac{100}{500}$

(6) Multi-programming brings multiple programs into memory. So they need to be protected from one another. Otherwise, to ensure correct execution, we need to save the memory contents, which would make multi-programing impractical.

(7) Although the total need is

300 + 200 + 300 = 800 MB
(A) (B) (C)

Each application is going to actually use much less. So we can fit them in memory much smaller than 800 MB.

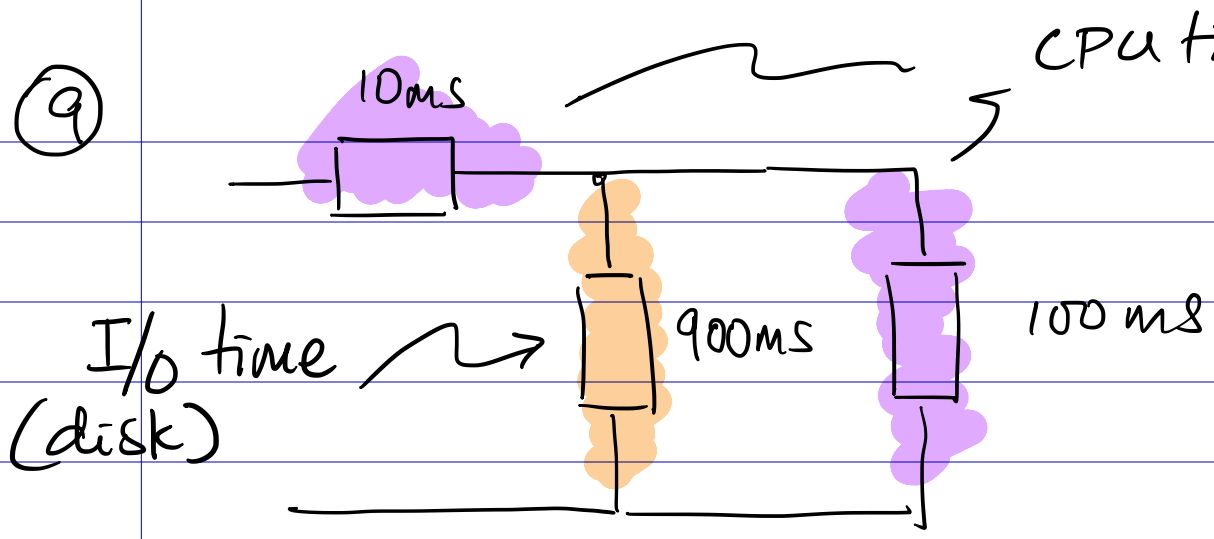
(8) Micro-kernel could be reliable because

(a) μkernel is very small so it can be reliably constructed.

(b) System does not crash unless μkernel crash. The services can crash and restart without affecting the system state as far is up or down

μkernel can make it less reliable because we need complex and evolving messaging interfaces.

(9)



Best scenario. is requests found in the cache.

One request served every 10 + 100 ms

Request rate = $1/0.110$ = 9.09 req/s

Worst case scenario, all requests go to disk.

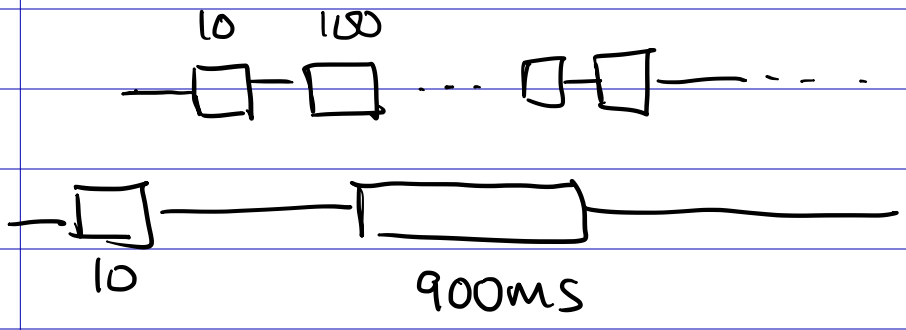
Request rate = $1/0.910$ = 1.09 req/s

(10) There is only one processor (standard assumption unless specified otherwise).

A compute activity cannot be overlapped with another compute activity.

We can overlap a disk I/O activity with CPU processing.

So the best we could do is to do back-to-back cache access. With multiple (two) threads, we can do the following.



910 ms we can complete 1+ 8 jobs.

Request rate $9 / 0.910 = 9.89$ req/s

(11) 2 threads

(12) 16 processes

(13) yield() {

= thread = get a pointer to the currently active thread

= Save the execution state (e.g., registers) in thread storage.

= lookup the ready threads and pick up the next available one to run

= load the execution state of newly picked thread

}

(14) Process switch does not need a system call.

With preemptive multiprocessing, we have a clock interrupt that triggers a process switch.

It happens in the kernel. A system call is used to go from user to kernel.

Overhead is 500 ns (interrupt servicing)
+ 200 ns (saving)
+ 200 ns (restoring)

900 ns in total.

(15) Message from A
Message from B

or in reverse order.

Outputs from both processes are there in the file

(16) Insert a close(1) before

printf ("Message from A.");

(17)
```
fd = dup(1)          // save stdout
close (1)
  =
  =
  -
if (fork() == 0) {
        close (1)
        dup (fd)     // restore stdout
```

This ensures the proper restoration of stdout.

(18) No

(19) So the program can access services offered by the kernel using the system call interface.

(20) Refer to lectures

(21) Refer to lectures

(22) Speedup is $\frac{600}{170} = 3.52$

(23) $s + p = 600$

$s + p/4 = 170$

$3p/4 = 430$

$p = 573.3$ s

$s = 26.7$ s

$\frac{573.3}{600}$ portion of the app. is parallel

(24) On a 16 core machine

$$S + P/16 = 26.7 + \frac{573.3}{16}$$

$$= 35.8 + 26.7$$

$$= 62.5\ s$$

(25) When you parallelize an application, each machine (processor) is running a smaller instance of the problem.

For example, an array used in computations could be much smaller.

⇒ They fit in the cache.

Better performance.

(26)

$$\text{Speedup} = \frac{1}{S + \frac{(1-S)}{N}}$$

S → Serial fraction of the application.

Lets say the overhead is O expressed as a fraction of the overall runtime.

$$= \frac{1}{S + \frac{(1-S)}{n} + O}$$

(27) The application developer was using a user-level threading library

(28) Faster context switches

Can create very large number of threads

(29) Does not block with blocking system calls

Can leverage multiple processing cores

(30) Large-scale simulations.

Event-driven frameworks that want to be lightweight.

Practice Questions #1

(1) How is a system call different from a function call?

(2) How is system call processing different from interrupt servicing?

(3) What is the significance of mode switching in system call processing? That is, can we have a system call without the mode switch?

(4) Consider a uni-programing system. The user wants to run job A and job B. A has the following characteristics: runs for 10 ms waits for 990 ms and the pattern continues for many iterations. Job B runs for 100ms and waits for 400ms and the pattern continues for many iterations. If you are measuring CPU utilization values over half second intervals, what are the possible values you could get?

(5) In the system given in (4), we enabled multi-programming. There is a single processor in the system and just jobs A and B. What are the time measured utilization values over half second intervals in this case?

(6) Why is memory protection an important concern with multi-programming?

(7) We have a computer with 512 MB of RAM. We have 3 applications. Application A needs 300 MB of RAM, B needs 200 MB, and C needs 300 MB. How can a virtual memory system load all three programs and still provide good performance? What characteristics of the applications are exploited by the virtual memory system?

(8) Can a micro-kernel OS be more reliable than a monolithic OS? Provide reasons why micro-kernel could provide a more reliable OS. Could micro-kernel make an OS less reliable too?

(9) Consider a web server. Each incoming request is processed as follows: input side processing (done on all incoming requests) – 10ms, disk access – 900ms, cache access – 100ms. The input side processing determines whether a request can be served by the cache or disk access is required. Assume that the web server has a single disk that serves the requests one after the other. The cache will hold popular requests after warm up. If the web server uses a single thread or process, what is the best and worst request rates achievable by the server?

(10) Suppose we use two threads in the above web server, what is the best request rate we could achieve?

(11) What is the maximum number of threads we can engage in the web server? That is find the number of threads beyond which the web server's request rate is not going to increase?

(12) Consider the following C code fragment. How many processes run soon after the execution of the given fragment?

```
for (i=0; i<4; i++) {
        fork();
        run_compute (i);
}
```

(13) You are expected to run programs in a computer with a sleeping beauty approach for resuming the dispatcher.

The support code provided by the professor for your assignment has a following core segment.

```
while (1) {
    obtainData (data)
    processData (data)
    outputData (data)
}
```

You immediately realize that there is a problem because the code does not relinquish the CPU and the program would not even yield the CPU for the OS. As a result, the OS would not be able to run and perform certain housekeeping tasks necessary for the code's execution.

```
while (1) {
    obtainData(data)
    processData(data)
    outputData(data)
}
```

yield();

You suggest the insertion of yield() as shown above.

What does yield() do? Can you provide the pseudo code for yield()?

(14) You are given a computing system where a system call takes 2000 ns (nano seconds), saving or restoring registers takes 200 ns, invoking interrupt servicing routine takes 500 ns. Approximately, how long would it take to switch from one process to another process?

(15) Consider the following code fragment.

```
close(1);
fd = open("temp2.txt", ...)
if (fork() == 0) {
        printf(" Message from A");
}
printf("Message from B");
```

What will be the contents of temp2.txt after executing this fragment?

(16) Lets say you don't want "Message from A" in your temp2.txt file. What modifications would you do?

(17) What modifications could you do to the code fragment to ensure that the output of "Message from A" shows up on the standard output?

(18) Can a program generate an address that is not in its address space?

(19) When a program is running in a modern operating system, portion of its address space is occupied by the kernel. Why is it necessary for the kernel to hold portion of the address space?

(20) Briefly explain how input redirection works in a UNIX like operating system. Show kernel-level data structures, system calls, and the order of manipulating them in in your answer.

(21) Briefly explain how command piping works in a UNIX like operating system. Show kernel-level data structures, system calls, and the order of their manipulations in your answer.

(22) An application took 600s to run in a single core machine. It took 170s on a four core machine. What is the speedup you got when runng it on the four core machine?

(23) What portion of the application is actually parallel?

(24) What is the expected runtine in a 16 core machine?

(25) Lets say the actual runtine you got when you run the application is less than the value computed above. Explain why you got better performance.

(26) Making a single application instance run over multiple cores involves additional programs like creating threads, distributing data, consolidating the results, etc. How can such overhead be factored into the speedup equation we saw in the class?

(27) You are asked to write a large program in C or C++ to solve a computational problem. You use a threading library because you realize that the application can be broken down to concurrent subtasks. By implementing the subtasks using threads you expect the application to run faster with more processor cores.

To your disappointment, your applicatn using threads actually runs slightly slower than the original serial version in a single core. You

explain it as threading overhead. With multiple cores, you are still at the same speed! What is going on? You go over the program and find no synchronization problems.

(28) What are the advantages of user-level threading over kernel level threading?

(29) What are the advantages of kernel level threading over user level threading?

(30) Can you think of an application that is better suited for user-level threading?