

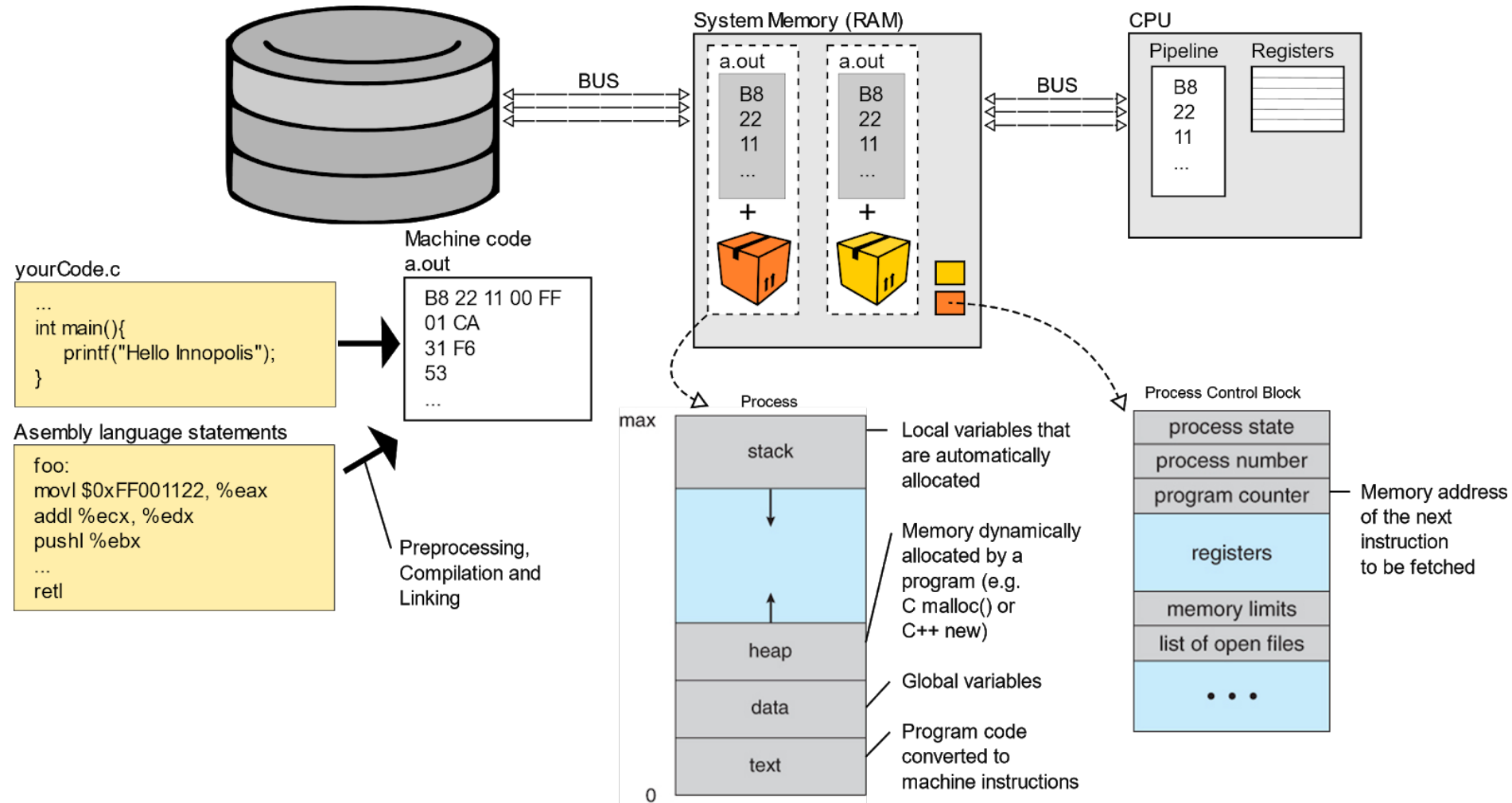
Interprocess Communication (IPC)

Week 05 - Tutorial

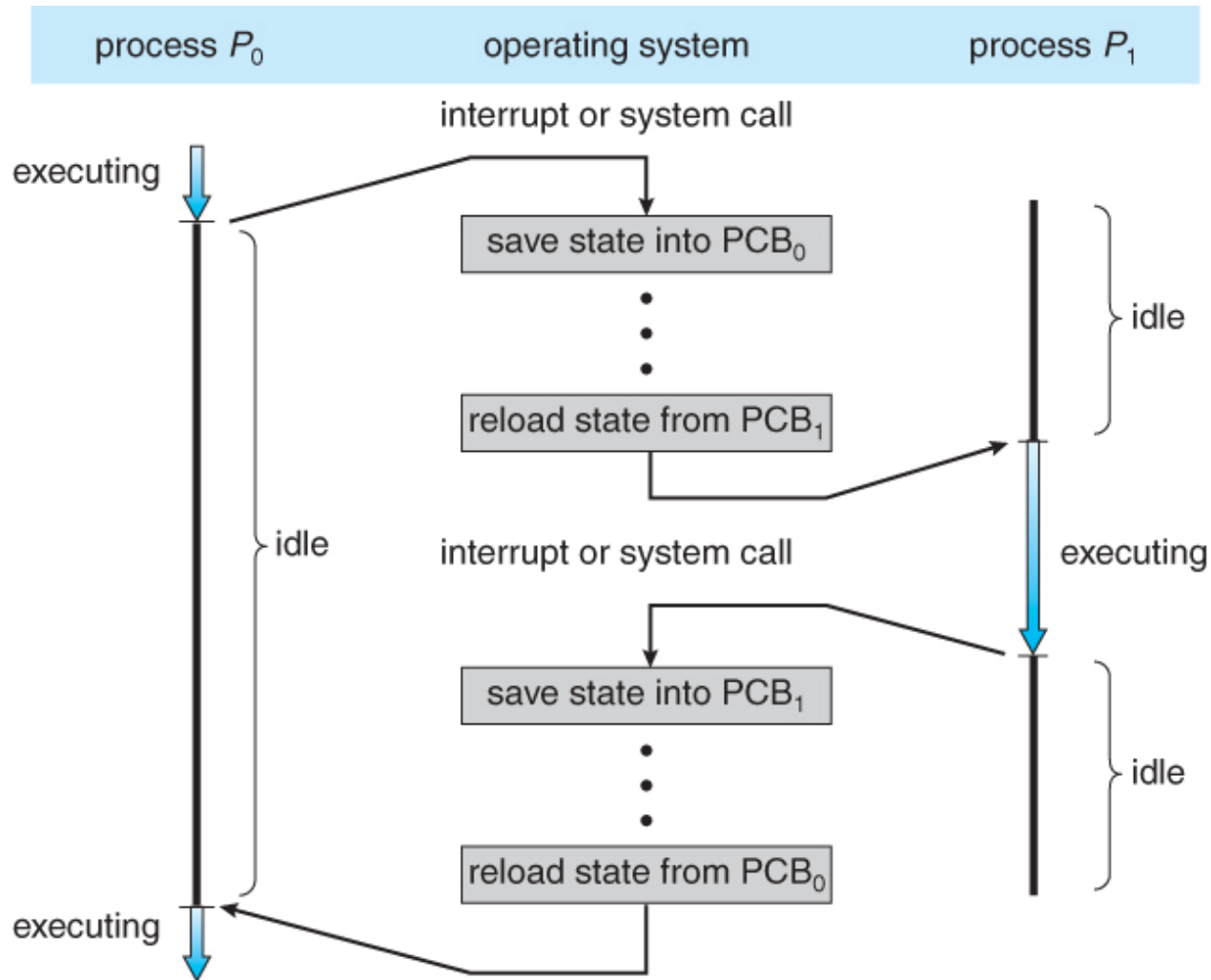
Outline

- Review of Processes and Threads
- Brief review on mutual exclusion
- Break
- Quiz

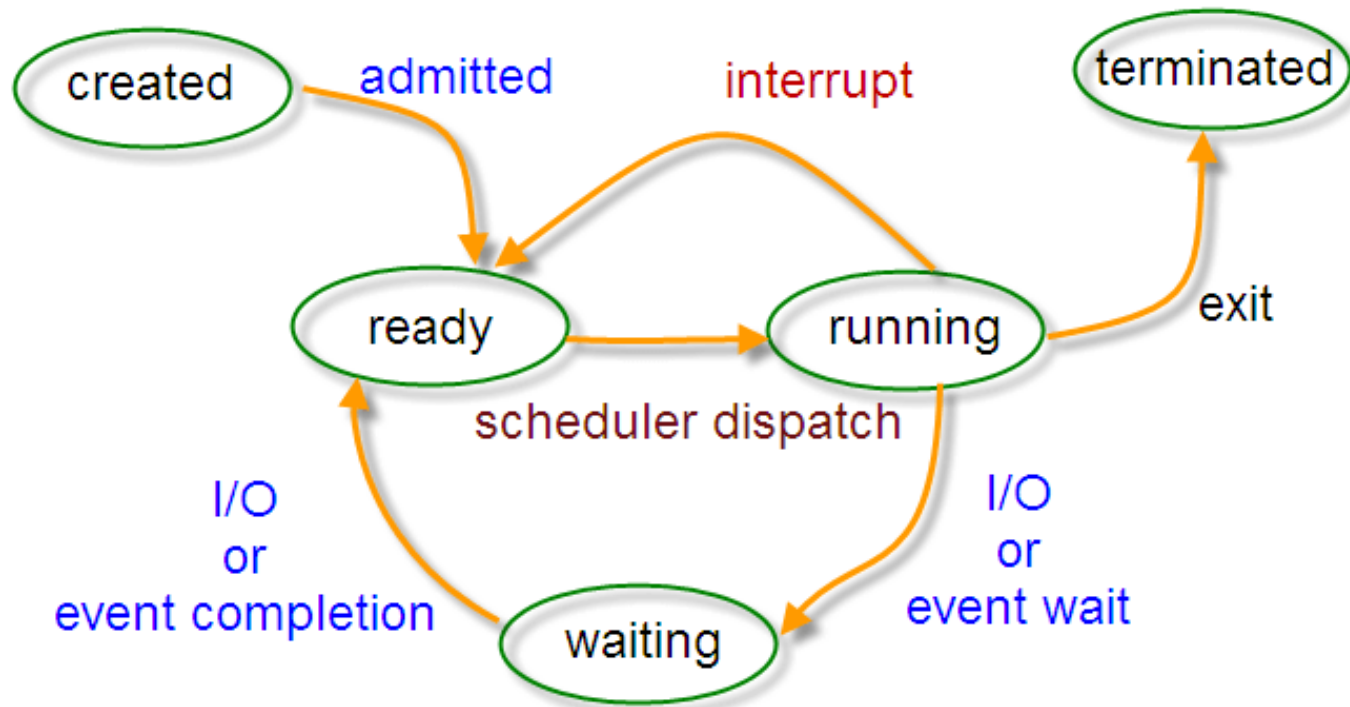
Process and Process Control Block



Context Switch



Process State



[States of process in Linux: shed.h](#)

Problem 2.8



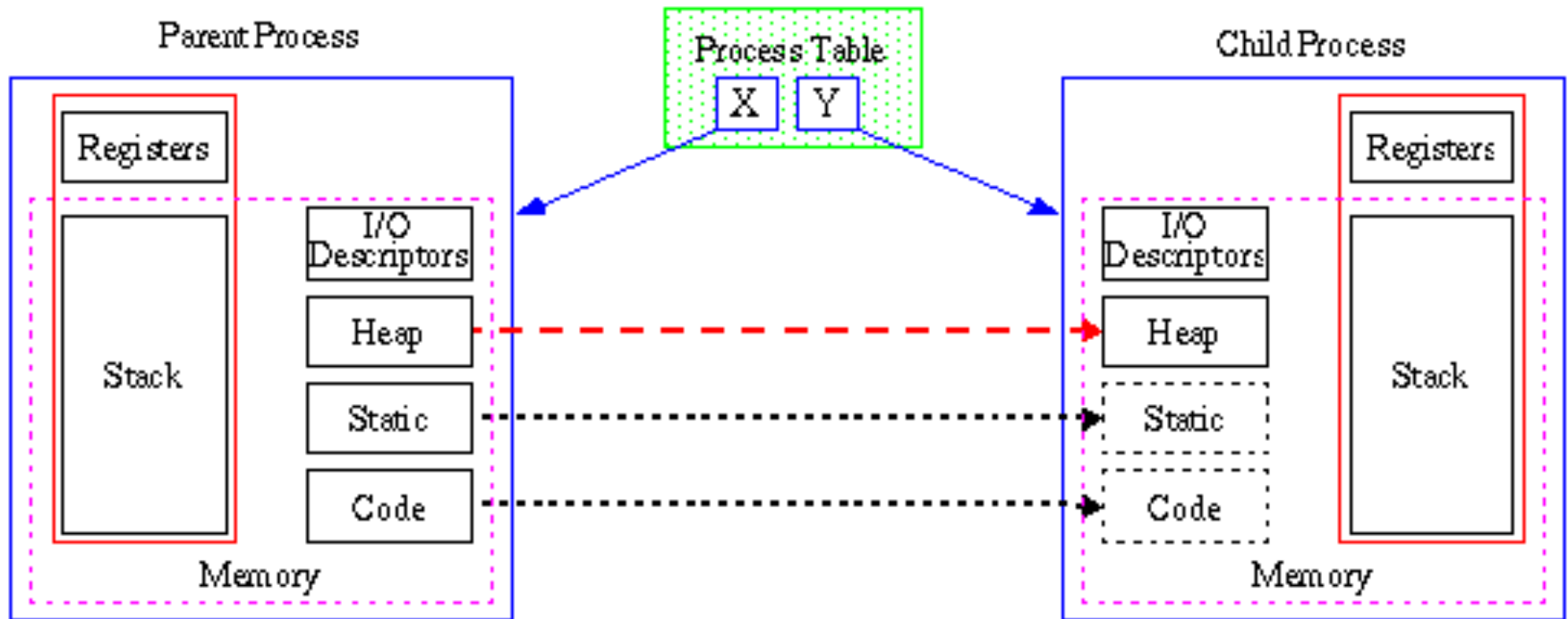
- Consider a multiprogrammed system with degree of 6 (i.e., six programs in memory at the same time).
- Assume that each process spends 40% of its time waiting for I/O.
- What will be the CPU utilization?

$$\text{CPU utilization} = 1 - p^n$$

Problem 2.8 - Solution

- Given that there are 6 programs in memory, $n=6$
- Each process spends 40% of time waiting for I/O, therefore the fraction of time each process spends waiting for I/O denoted by $P = 0.4$
- CPU utilization is given as $= 1 - P^n = 1 - (0.4)^6 = 1 - 0.004096 = 0.995904$
- Therefore, the CPU utilization is 99.59%

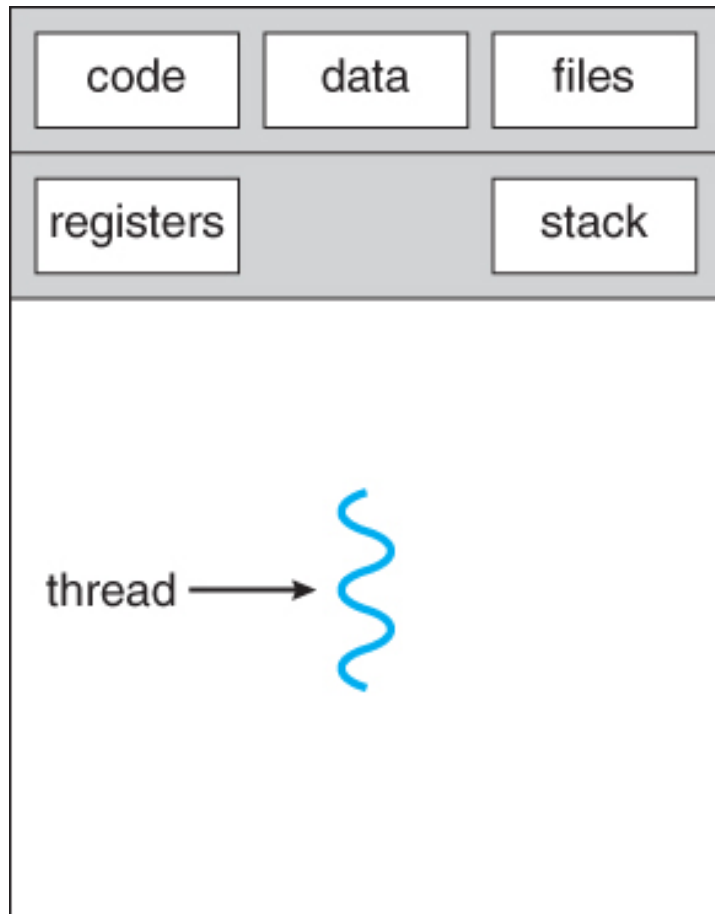
Fork



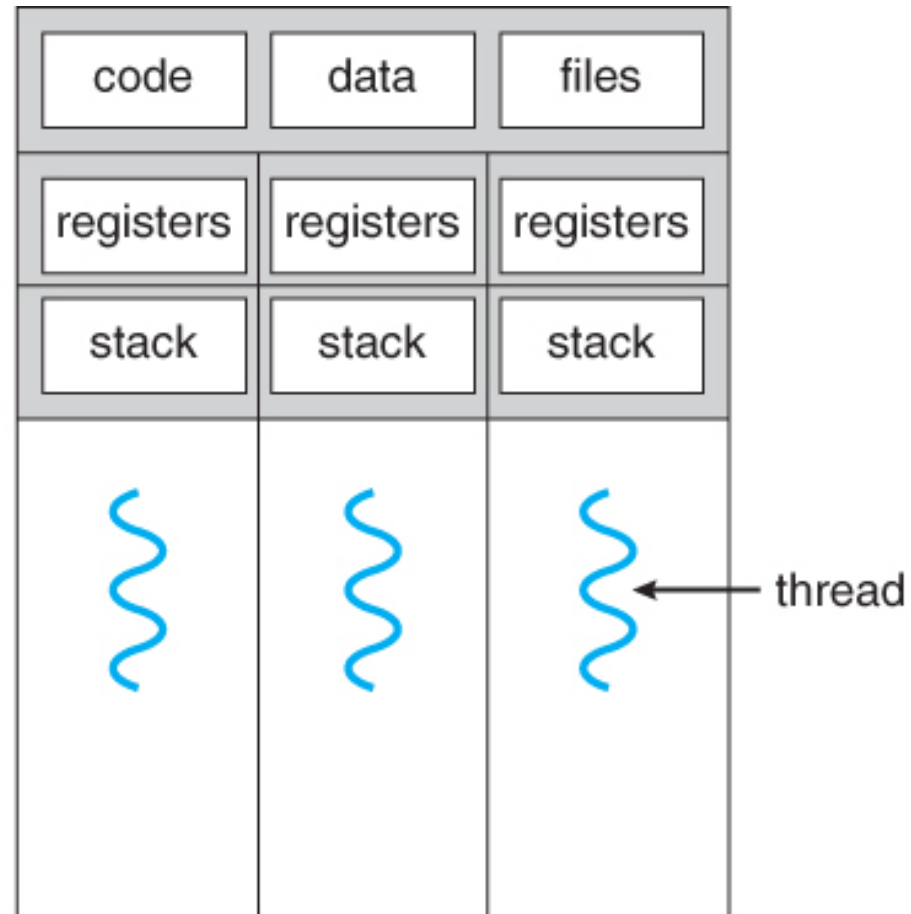
“Fork is the only way to create a new process in POSIX (standard for UNIX that most versions of UNIX support). It creates an exact duplicate of the original process, including all the file descriptors, registers - everything. After the fork, the original process and the copy (the parent and child) go their separate ways.”

Modern Operating Systems - Tanenbaum and Bos

Processes and Threads

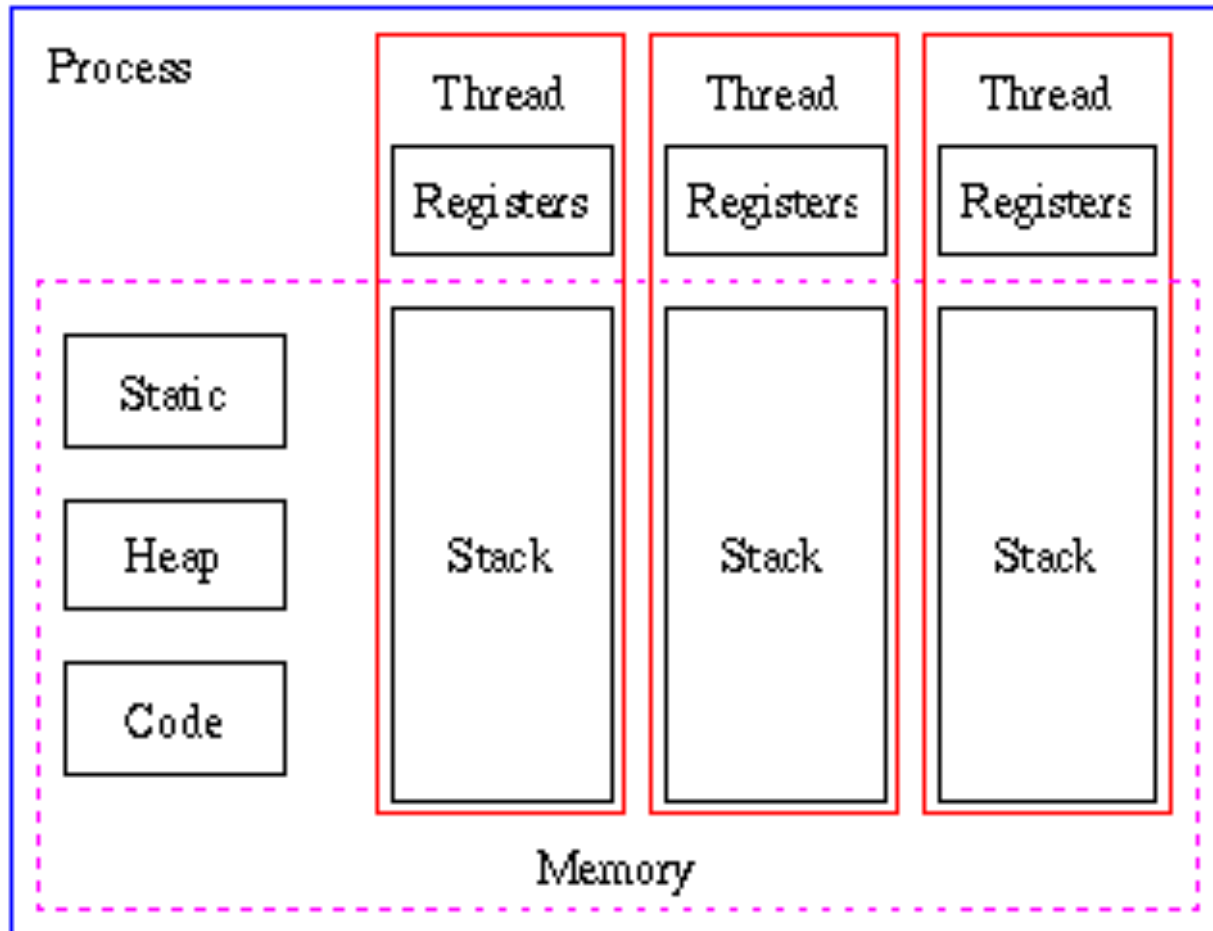


single-threaded process



multithreaded process

Threads in a process



Problem 2.14



In the table the register set is listed as a per-thread rather than a per-process item.
Why? After all, the machine has only one set of registers.

Per-process items	Per-thread items
Address space	Program counter
Global variables	Registers
Open files	Stack
Child processes	State
Pending alarms	
Signals and signal handlers	
Accounting information	

Problem 2.14 - Solution

- The register is called a per-thread item because the context saved in a register is thread-specific information
- The register stores the state of every thread so that it can be used during context switching - these data are saved and reloaded on the next execution

Mutual exclusion, Busy waiting



www.alamy.com - BXM0HD

Problem 2.30



- Consider the following solution to the mutual-exclusion problem involving two processes P0 and P1. Assume that the variable turn is initialized to 0. Process P0's code is presented below:

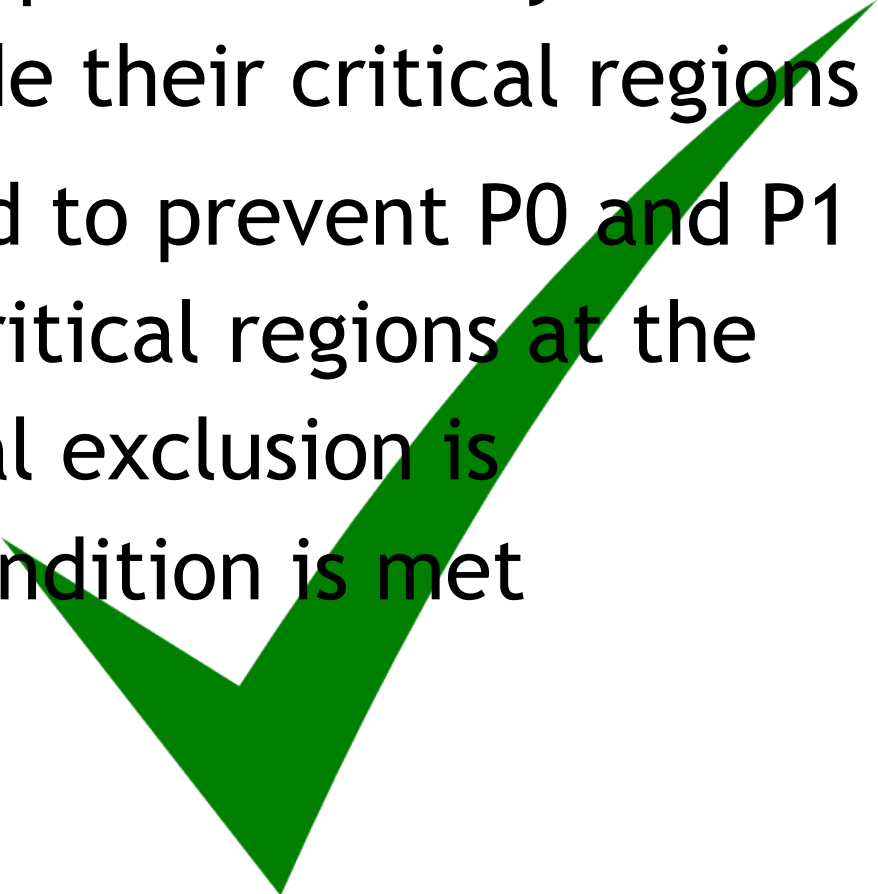
```
/* Some code */  
while (turn != 0) { } /* Do nothing and wait. */  
  
Critical Section /* . . . */  
turn = 0;  
/* Some other code */
```

- For process P1, replace 0 by 1 in above code. Determine if the solution meets all the required conditions for a correct mutual-exclusion solution

Problem 2.30 - Solution (1 / 4)

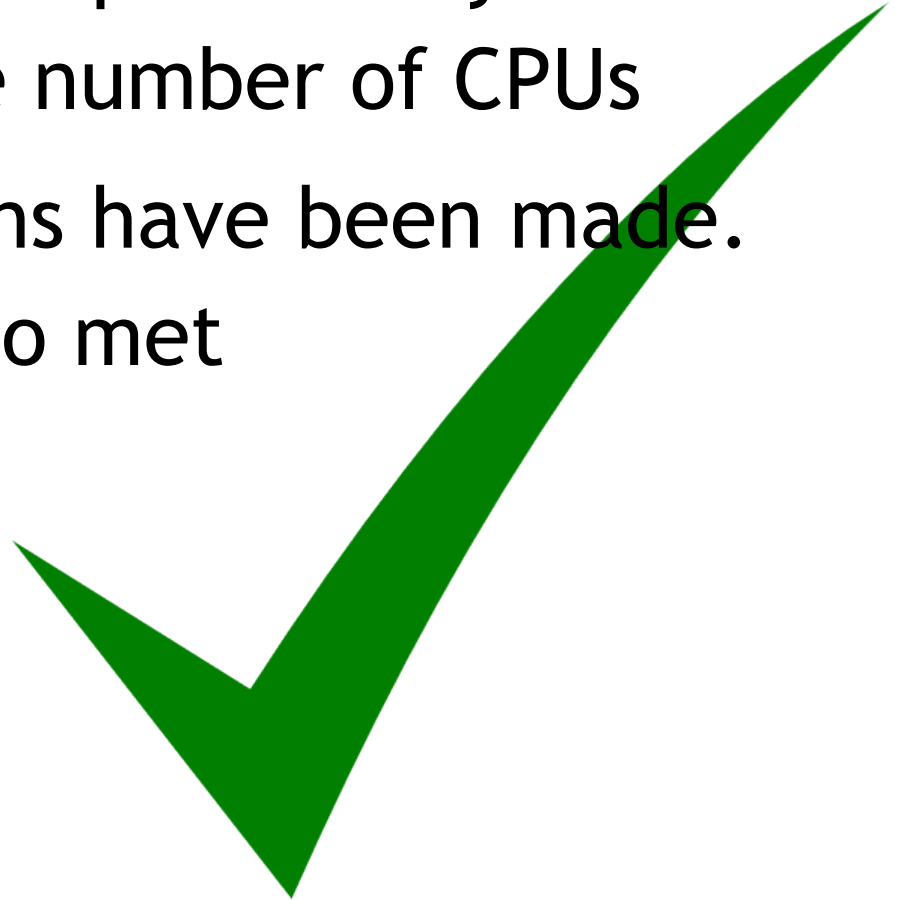
- First, remember four required conditions:
 1. No two processes may be simultaneously inside their critical regions
 2. No assumptions may be made about speeds or the number of CPUs
 3. No process running outside its critical region may block other processes
 4. No process should have to wait forever to enter its critical region

Problem 2.30 - Solution (2/4)

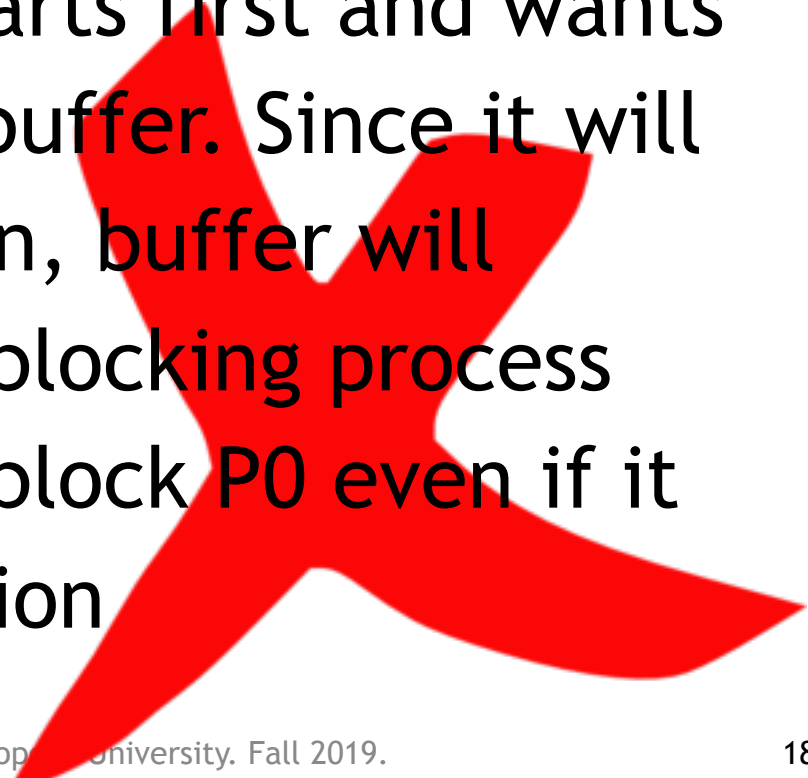
- Condition 1: no two processes may be simultaneously inside their critical regions
 - Variable *turn* is used to prevent P0 and P1 being inside their critical regions at the same time so mutual exclusion is guaranteed. This condition is met
- 

Problem 2.30 - Solution (3/4)

- Condition 2: no assumptions may be made about speeds or the number of CPUs
- Okay, no assumptions have been made.
This condition is also met



Problem 2.30 - Solution (4/4)

- Condition 3: no process running outside its critical region may block other processes
 - Suppose, process P1 starts first and wants to put something into buffer. Since it will not enter critical region, buffer will remain empty causing blocking process P0. Therefore, P1 will block P0 even if it is not in its critical region
- 

Problem 2.11

If a multithreaded process forks, a problem occurs if the child gets copies of all the parent's threads. Suppose that one of the original threads was waiting for keyboard input. Now two threads are waiting for keyboard input, one in each process. Does this problem ever occur in single-threaded processes?

Problem 2.11 - Solution

If a process define a signal handler or blocked to get user input and then forked - every process will get the signal* **

*it's tricky and has a lot of pitfalls and undefined behaviour conditions

** but not the input, the input may be empty, since the memory will be read)

In a single-threaded system there's no such a problem - you first block, then fork.

Problem 2.16

Can a thread ever be preempted by a clock interrupt? If so, under what circumstances? If not, why not?

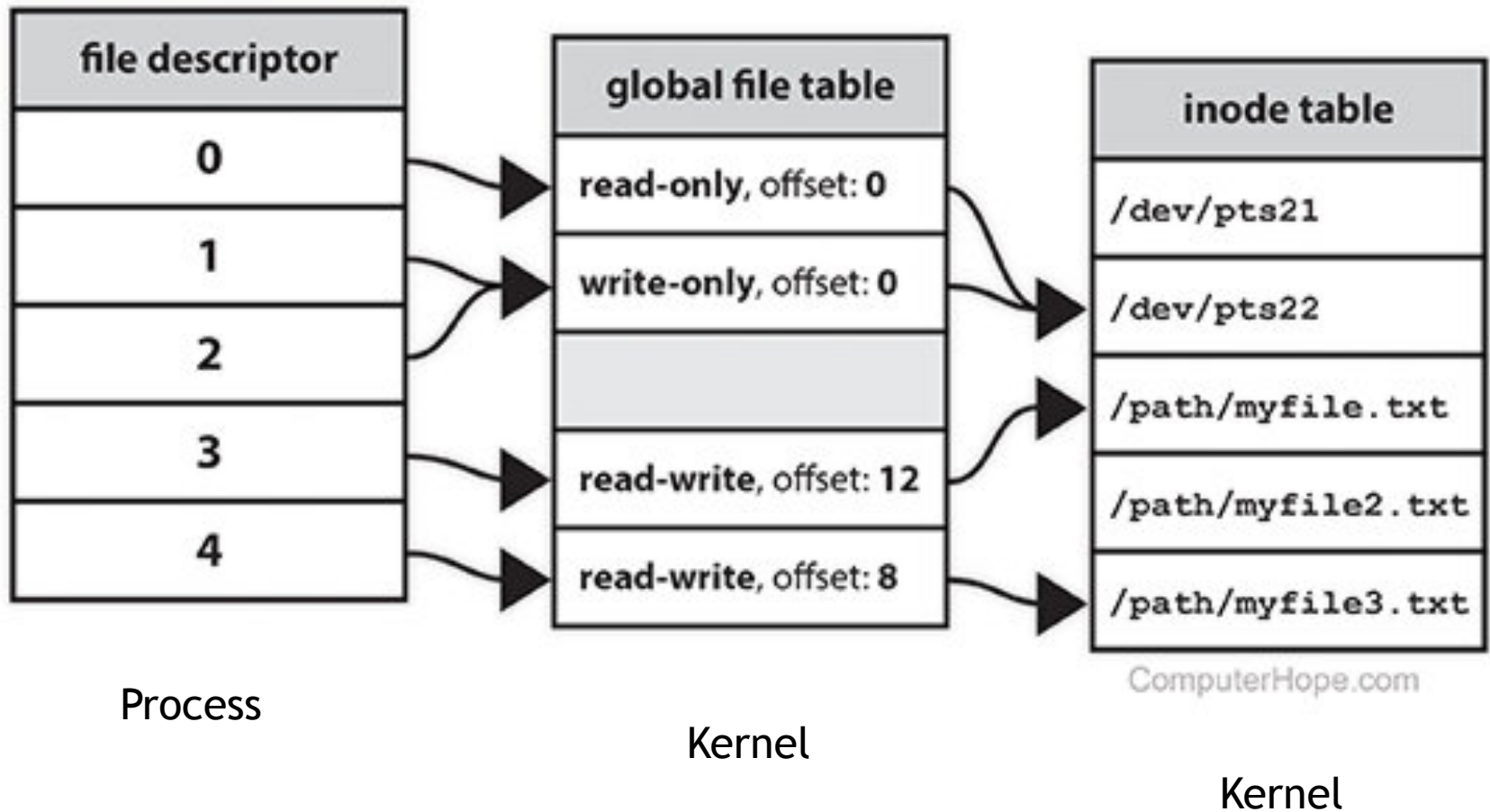
Problem 2.16

- A user level thread cannot be preempted by the clock in a sense that a clock preempts processes
- A kernel level thread can be preempted since the scheduler knows of their existence

Signals

- `$ man signal`
- **How to send a signal?** `$ man 2 kill`
 - `kill(pid_t target_pid, int signal);`
- **How to receive a signal?** `$ man 2 sigaction`
 - `sigaction(int signal, ..., void *handler);`

File Descriptors



References

- <http://www.sanfoundry.com/operating-system-questions-answers-basics/>

End

Week 05 - Tutorial