Chapter 2 - Processes and Threads

Luis Tarrataca
luis.tarrataca@gmail.com

CEFET-RJ

1 Motivation

2 Processes

The process model

Process Termination

Process States

Process States

Scheduling Techniques

Linux Scheduler Example

3 Threads

- Classical Thread Model
- POSIX threads
- Threads in User Space
- Threads in the Kernel
- Hybrid Implementations
- Making Single Threaded Code Multithreaded

4 Interprocess Communication

Race Conditions

Critical Regions

Mutual Exclusion with Busy Waiting

Disabling interrupts

Lock Variables

Strict Alternation

TSL Instruction

Sleep and Wakeup

Semaphores

Mutexes

Mutexes in Pthreads

Monitors

5 Scheduling

When to schedule

Scheduling Algorithm Goals

Scheduling

6 Classical IPC Problems

The Dining Philosophers Problem

Readers and Writers Problem

Motivation

From the previous slides:

> What is one of the most important concepts in OS? Any ideas?

Motivation

From the previous slides:

What is one of the most important concepts in OS? Any ideas?

- Process

Motivation

From the previous slides:

What is one of the most important concepts in OS? Any ideas?

- Process

Do you remember what a process is?

Motivation

Do you remember what a process is?

- Abstraction of a running program;
- *I.e.* the state of a program:
  - PSW (PC, IR, ...);
  - Files opened;
  - Sockets used;
  - Every resource being used by the program;
- One of the oldest and most important abstractions;
- Turn a single CPU into multiple virtual CPUs;

Processes

Modern computers do several things simultaneously:

- Check emails, run text editor, play music, etc...
- This was not always like this:
  - Computers used to be able to run a single program;
- Clearly some mechanism is need to model and control this **concurrency**:
  - Share the resource among many programs:
    - CPU;
    - Disk;
    - Etc...

Clearly some mechanism is need to model and control this **concurrency**:

Do you have any idea of what this concept is?

Clearly some mechanism is need to model and control this **concurrency**:

Do you have any idea of what this concept is?

- Processes ;)
- And also **threads**
  - Which we will see later;

In any **multiprogramming environment** (1/2):

- CPU switches from process to process quickly;
  - Each runs for a duration of time;
  - Determined by some algorithm;
- **Scheduler** process is responsible for:
  - Changing processes;
  - Deciding who to run next;
  - In Portuguese:
    - escalonador / agendador

In any **multiprogramming environment** (2/2):

- Each process typically runs tens / hundreds of miliseconds:
  - In one second several processes will have been executed;
- This gives the illusion of parallelism:
  - **pseudoparallelism**
- This contrasts with true multiprocessor parallelism;

The process model

In essence, in a **multiprogramming environment**:

- OS have process concept;
- OS alternates process execution;

We will assume there is only one CPU:

- In reality multiple cores will exist;
- But if we know how to process one core:
  - Easy to extend for all cores;
  - Each core will only run a process at a time;

Multiprogramming idea:

- Make efficient use of the processor;
- If we only have a single process executing:
  - Eventually some I/O operation will need to be performed;
  - Program will have to wait for the result of this operation:
    - I/O operations much slower than CPU
  - Processor will be **idle** a long time;

Figure: Executing a single program (Source: (Stallings, 2015))

Figure: Executing a single program (Source: (Stallings, 2015))

Is this an efficient use of the processor?

Most of the time the processor is idle not doing anything.

- Processor executes orders of magnitude faster than I/O...
- Consider the following example:

Read one record from file	15 μs
Execute 100 instructions	1 μs
Write one record to file	15 μs
TOTAL	31 μs

$$\text{Percent CPU utilization} = \frac{1}{31} = 0.032 = 3.2\%$$

Figure: System utilisation Example (Source: (Stallings, 2015))

Instead of idling the system we could be running another program...

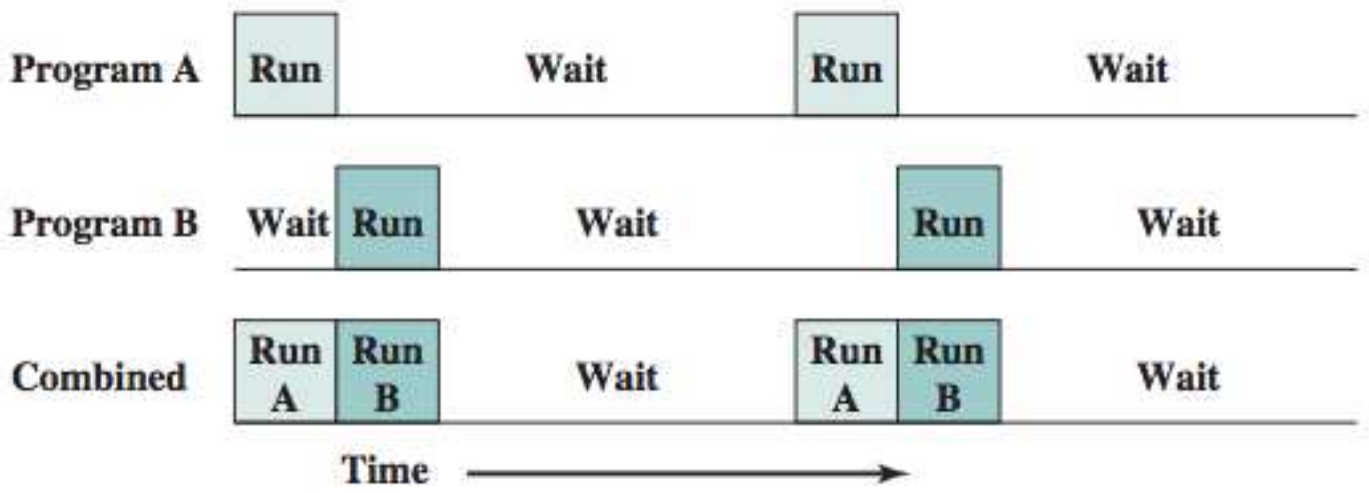

Figure: Executing two programs (Source: (Stallings, 2015))

But this second program may eventually also ask for I/Os...

We can even add a third program...

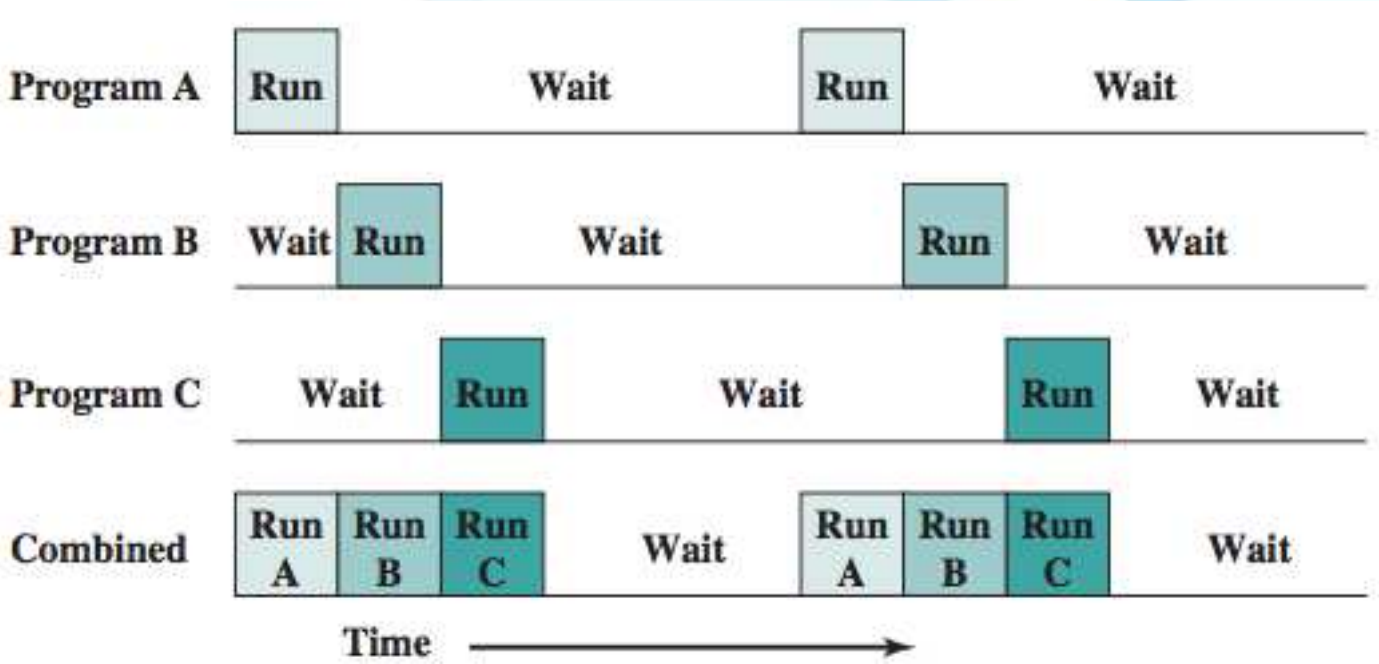

Figure: Executing three programs (Source: (Stallings, 2015))

This way the processor idle times are diminished...

CPU switches back and forth between processes:

- Process computation will not be uniformly executed:
  - Some code sections may run for a longer time than others;
  - This depends on resource competition at any given time:
- **Therefore:**
  - Processes must **not** be programmed with built-in assumptions about timing;
  - If a timer is need OS have timer system calls available:
    - *E.g.:* setitimer, alarm

Example

Assume the following:

- A process spends a fraction p of its time waiting for I/O;
- Simplification: assume n processes spend the same time waiting for I/O
  - Probability that all n processes are waiting for I/O is

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

Example

We can then plot the following picture

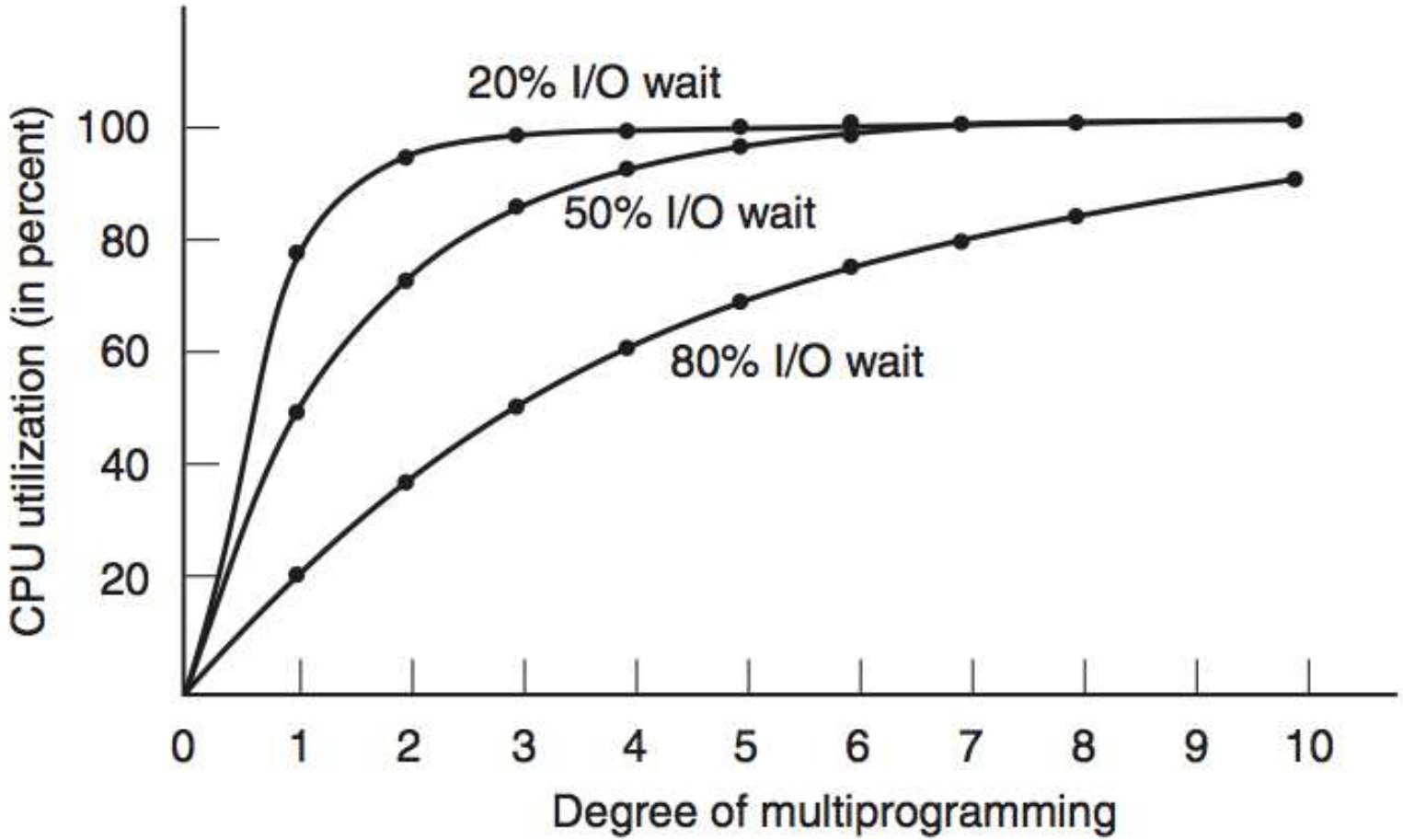

Figure: CPU utilization as a function of the number of processes in memory. (Source: (Tanenbaum and Bos, 2015))

Example

From the previous figure:

- If processes spend 80% of their time waiting for I/O:
  - At least 10 processes must be in memory for CPU waste to fall below 10%;
- If processes spend 50% of their time waiting for I/O:
  - At least 4 processes must be in memory for CPU waste to fall below 10%;
- If processes spend 20% of their time waiting for I/O:
  - At least 3 processes must be in memory for CPU waste to fall below 10%;

In reality:

- $\neq$ processes will have $\neq$ I/O times;

What do you think is the main conclusion to draw from the previous slides? Any ideas?

In reality:

- $\neq$ processes will have $\neq$ I/O times;

What do you think is the main conclusion to draw from the previous slides? Any ideas?

- CPU Utilization rate should be close to 100%;
- Only possible with high degree of multiprogramming;

In UNIX, it is possible to see processes through several commands:

- ps
- top
- htop

Which will show a listing of the processes:

- Most of them will be **daemons**:
- *I.e.:* process running in background to handle some activity;

Process listing through *htop*:

```
  1  [||||||||||                                                  ]   Tasks: 226, 668 thr; 1 running
  2  [|                                                           ]   Load average: 1.74 1.47 1.44
  3  [||||||                                                      ]   Uptime: 02:52:38
  4  [|||                                                         ]
  Mem[|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||]
  Swp[                                                            ]

  PID USER      PRI  NI  VIRT   RES S CPU% MEM%   TIME+  Command
  944 Taka       24   0 3751M  546M ?  8.6  3.3 10:03.29 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  639 Taka       32   0 3712M  324M ?  2.2  2.0 12:32.65 /Applications/Internet & Networking/Google Chrome.app/Contents/MacOS/Google Chrome
  912 Taka       24   0 2528M 84204 ?  1.5  0.5  0:01.89 /System/Library/PrivateFrameworks/HelpData.framework/Versions/A/Resources/helpd
  789 Taka       24   0 2594M 36240 ?  1.4  0.2  0:02.36 /Applications/Utilities/Terminal.app/Contents/MacOS/Terminal
  661 Taka       24   0 3966M  748M ?  1.0  4.6  3:15.73 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  269 Taka       17   0 2421M  8404 ?  0.5  0.1  0:06.98 /usr/sbin/distnoted agent
  451 Taka       24   0 4267M  205M ?  0.4  1.3  3:56.99 /Users/Taka/Library/Application Support/Dropbox/Dropbox.app/Contents/MacOS/Dropbox /newerversion
 1198 Taka       24   0 2395M  2248 R  0.3  0.0  0:00.03 htop
 1128 Taka       17   0 2474M 35080 ?  0.1  0.2  0:01.02 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c M
 1138 Taka       17   0 2459M 22748 ?  0.1  0.1  0:00.96 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c M
  403 Taka       40   0 2476M 17096 ?  0.0  0.1  0:04.23 /usr/libexec/SafariCloudHistoryPushAgent
  297 Taka       17   0 2579M 40628 ?  0.0  0.2  0:04.83 /System/Library/CoreServices/Spotlight.app/Contents/MacOS/Spotlight
  283 Taka       17   0 2516M 32648 ?  0.0  0.2  0:07.35 /System/Library/CoreServices/Dock.app/Contents/MacOS/Dock
  421 Taka       32   0  664M  5188 ?  0.0  0.0  0:00.92 /Library/Printers/Brother/Utilities/Server/NETserver.app/Contents/MacOS/NETserver
 1196 Taka       17   0 3212M 55172 ?  0.0  0.3  0:00.43 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  299 Taka       17   0 2470M  9844 ?  0.0  0.1  0:06.40 /System/Library/Frameworks/ApplicationServices.framework/Frameworks/ATS.framework/Support/fontd
  650 Taka       17   0 3433M  273M ?  0.0  1.7  0:35.94 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  271 Taka       17   0 2533M  121M ?  0.0  0.7  0:04.75 /usr/sbin/cfprefsd agent
 1197 Taka       17   0 3237M 80252 ?  0.0  0.5  0:00.93 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  495 Taka       16   0  335M   560 ?  0.0  0.0  0:01.57 /Library/DropboxHelperTools/Dropbox_u502/dbfseventsd
  649 Taka       17   0 3192M 50980 ?  0.0  0.3  0:00.79 /Applications/Internet & Networking/Google Chrome.app/Contents/Versions/59.0.3071.115/Google Chrome Helper.app/Contents/MacO
  397 Taka       17   0 3527M 17224 ?  0.0  0.1  0:00.35 /Applications/Diverse/Spectacle.app/Contents/MacOS/Spectacle
  285 Taka       17   0 2514M 21076 ?  0.0  0.1  0:01.58 /System/Library/CoreServices/SystemUIServer.app/Contents/MacOS/SystemUIServer
 1175 Taka       32   0 2394M  1312 ?  0.0  0.0  0:00.00 -bash
 1174            32   0     0     0 ?  0.0  0.0  0:00.00 login
 1168            17   0     0     0 ?  0.0  0.0  0:00.00 ocspd
 1139 Taka       17   0 2421M  7644 ?  0.0  0.0  0:00.11 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker-lsb
 1137 Taka       17   0 2421M  7728 ?  0.0  0.0  0:00.18 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker-lsb
 1131 Taka       17   0 2445M  7888 ?  0.0  0.0  0:00.23 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -c M
 1127 Taka       17   0 2460M 21708 ?  0.0  0.1  0:01.05 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -c M
 1106 Taka       17   0 2432M 19712 ?  0.0  0.1  0:00.35 /System/Library/Frameworks/CoreServices.framework/Frameworks/Metadata.framework/Versions/A/Support/mdworker -s mdworker -c M
 1104            40   0     0     0 ?  0.0  0.0  0:00.00 tccd
 1103 Taka       17   0 2412M  6644 ?  0.0  0.0  0:00.03 /System/Library/PrivateFrameworks/CommerceKit.framework/Versions/A/XPCServices/com.apple.CommerceKit.TransactionService.xpc/
 1101            17   0     0     0 ?  0.0  0.0  0:00.00 AssetCacheLocato
 1099            17   0     0     0 ?  0.0  0.0  0:00.00 softwareupdate_d
 1098            17   0     0     0 ?  0.0  0.0  0:00.00 installd
 1015            40   1     0     0 ?  0.0  0.0  0:00.00 periodic-wrapper
  980            17   0     0     0 ?  0.0  0.0  0:00.00 nbstated
  979 Taka       24   0 2487M 14668 ?  0.0  0.1  0:00.36 /System/Library/PrivateFrameworks/Noticeboard.framework/Versions/A/Resources/nbagent.app/Contents/MacOS/nbagent
  919 Taka       24   0 2439M 14128 ?  0.0  0.1  0:00.46 /System/Library/PrivateFrameworks/CloudServices.framework/Versions/A/XPCServices/com.apple.lakitu.xpc/Contents/MacOS/com.app
  916 Taka       17   0 2436M  7484 ?  0.0  0.0  0:00.08 /System/Library/PrivateFrameworks/CloudServices.framework/Resources/com.apple.sbd
  915 Taka       17   0 2488M 16944 ?  0.0  0.1  0:00.17 /System/Library/PrivateFrameworks/CloudServices.framework/Resources/EscrowSecurityAlert.app/Contents/MacOS/EscrowSecurityAle
  800 Taka       17   0 2573M 47764 ?  0.0  0.3  0:07.86 /System/Library/Frameworks/Quartz.framework/Frameworks/QuickLookUI.framework/Resources/QuickLookUIHelper.app/Contents/MacOS/
  792 Taka       25   0 2395M  1340 ?  0.0  0.0  0:00.02 -bash
  791            32   0     0     0 ?  0.0  0.0  0:00.00 login
  747 Taka       40   0 2435M  9320 ?  0.0  0.1  0:00.03 /System/Library/PrivateFrameworks/BookKit.framework/Versions/A/XPCServices/com.apple.BKAgentService.xpc/Contents/MacOS/com.a
F1Help  F2Setup F3SearchF4FilterF5Tree  F6SortByF7Nice -F8Nice +F9Kill  F10Quit
```

A running process may issue system calls to create new processes:

- In UNIX: *fork* system call;
- Useful if work can be performed by other processes;
- No need to develop additional code;
- Capitalize on existing knowledge: Save time on bugs;
  - In UNIX: *execv* system call;

> But you want to know the real reason why this is done?

A running process may issue system calls to create new processes:

- In UNIX: *fork* system call
- Useful if work can be performed by other processes;
- No need to develop specific code;
- Reutilize everything that was already made:
  - Capitalize on existing knowledge;
  - Save time on bugs;

> But you want to know the real reason why this is done?

  - Programmers are lazy ;)

After a process is created:

- parent and child have their own distinct address spaces;
- If either process changes a word in its address space:
  - Change is not visible to the other process;

Process Termination

After a process has been created:

- It starts running and does whatever its job is.
- Eventually the process will terminate, usually due to one of the following:
  - Normal exit (voluntary).
  - Error exit (voluntary).
  - Fatal error (involuntary).
  - Killed by another process (involuntary).
- Most processes terminate because they have done their work:
  - In UNIX: *exit* system call

Process States (1/2)

During the lifetime of a process, its state will change a number of times:

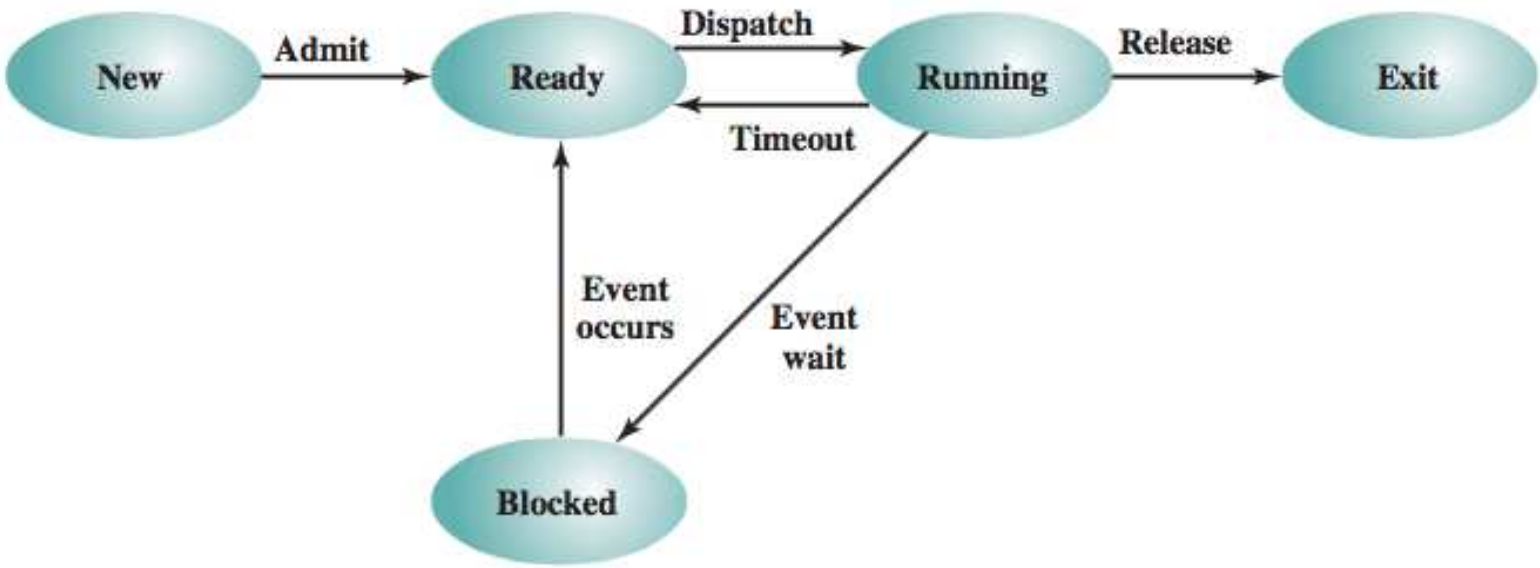

Figure: Five state process model (Source: (Stallings, 2015))

Process States (2/2)

During the lifetime of a process, its state will change a number of times:

- **New:** Process is created but not yet ready to execute.
- **Ready:** Process is ready to execute, awaiting processor availability;
- **Running:** Process is being executed by the processor;
- **Waiting:** Process is suspended from execution waiting a system resource;
- **Halted:** Process has terminated and will be destroyed by the OS.

Process Control Block

OS represents each process by a **control block** (simplified) (1/2):

- **Identifier:** Unique process identifier;
- **State:** Current process state;
- **Priority:** Process priority level.;
- **Program counter:** Next instruction;
- **Memory pointers:** Process starting and ending memory locations;
- **Context data:** Processor state registers;
- **I/O status:** I/O requests and I/O devices;
- **Accounting Info:** *E.g.* processor time, clock time, time limits,...

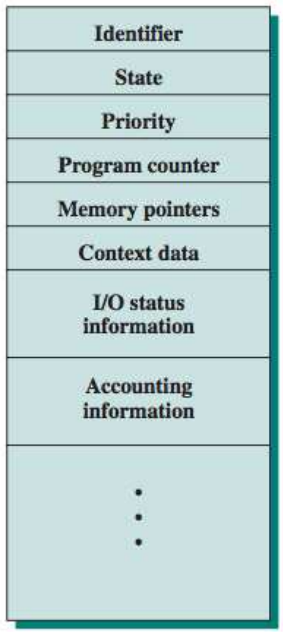

Figure: Process Control Block (Source: (Stallings, 2015))

OS represents each process by a **control block** (detailed) (2/2):

Process management	**Memory management**	**File management**
Registers	Pointer to text segment info	Root directory
Program counter	Pointer to data segment info	Working directory
Program status word	Pointer to stack segment info	File descriptors
Stack pointer		User ID
Process state		Group ID
Priority		
Scheduling parameters		
Process ID		
Parent process		
Process group		
Signals		
Time when process started		
CPU time used		
Children's CPU time		
Time of next alarm		

Figure: Scheduling Example (Source: (Stallings, 2015))

Scheduling Techniques

Consider the following scenario:

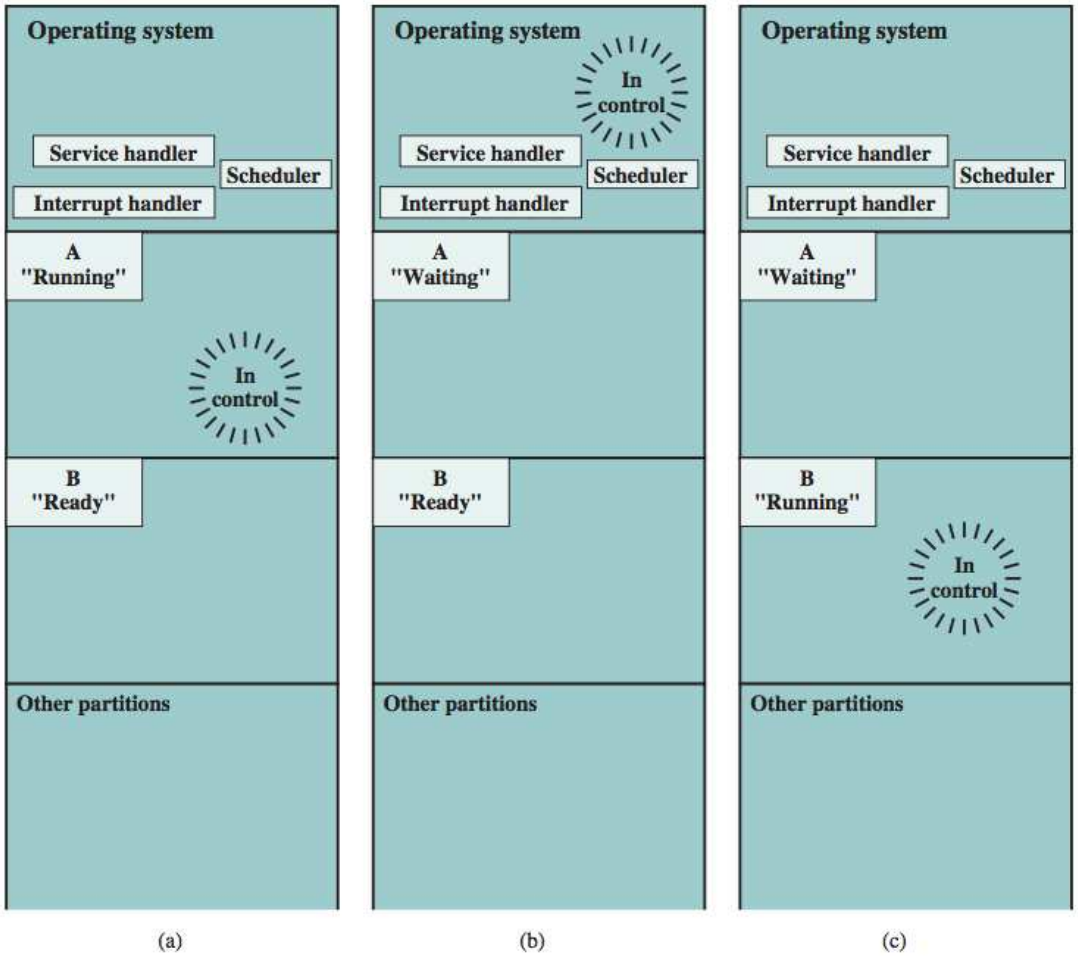

Figure: Scheduling Example (Source: (Stallings, 2015))

Initially process **A** is running and:

1. The processor is executing instructions from process **A**;
2. The processor then:
   - ceases to execute **A**;
   - begins executing OS instructions.
3. This will happen for one of three reasons:
   1. Process A issues a service call (e.g., an I/O request) to the OS.
      - Execution of A is suspended until this call is satisfied by the OS.
   2. Process A causes an interrupt signal:
      - When this signal is detected, the processor ceases to execute A;
      - OS processes the interrupt signal;
   3. An event unrelated to process A causes an interrupt.
      - *E.g.* is the completion of an I/O operation.

Process **A** therefore is going to block and control is passed to the OS:

1. The OS saves:
   - Current processor context (registers);
   - PC;
2. The OS:
   1. changes the state of **A** to **blocked**;
   2. decides which process should be executed next;
   3. instructs the processor to restore B's context data;
   4. proceeds with the execution of B where it left off.

This gives rise to the following model:

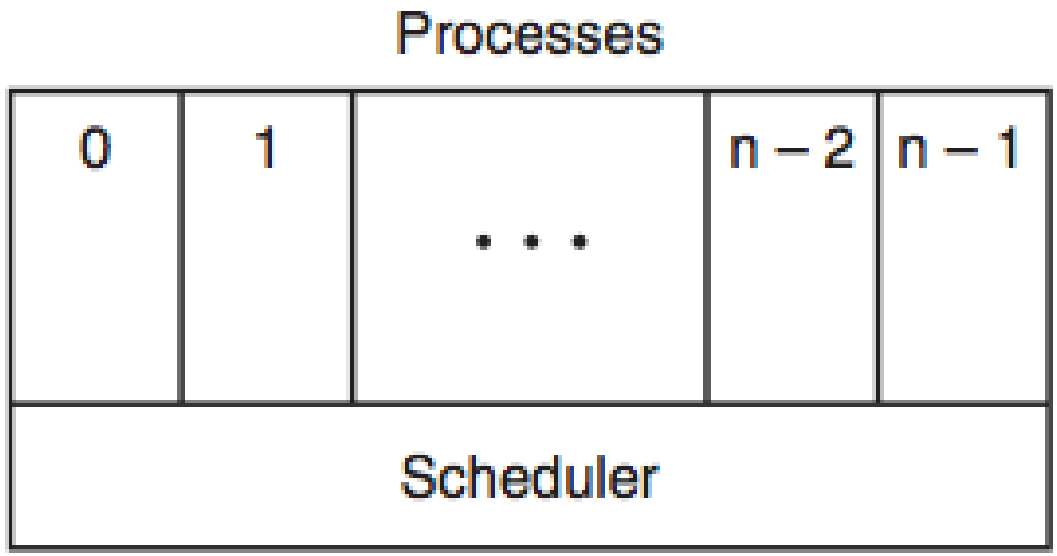

Figure: The lowest layer of a process-structured operating system handles interrupts and scheduling. Above that layer are sequential processes. (Source: (Tanenbaum and Bos, 2015))

- OS lowest level is the **scheduler**;

How does the scheduler choose among the various processes? Any ideas?

How does the scheduler choose among the various processes?

Well it depends on a lot of variables:

- **Real-time operating systems:** is the OS responsible for:
  - Controlling a nuclear station?
  - Controlling an airplane?
- **General purpose operating systems:**
  - Do different processes have different priorities?
  - How long has a process been allowed to run?
  - ...

So, let me ask the question again:

> How does the scheduler choose among the various processes? Any ideas?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

- Array?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

- Array?
- List?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

- Array?
- List?
- Queue?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

- Array?
- List?
- Queue?
- Hash table?

So, let me ask the question again:

How does the scheduler choose among the various processes? Any ideas?

What would be a good way to make such a choice in a computational manner? Any ideas?

- Array?
- List?
- Queue?
- Hash table?
- Tree?

Linux Scheduler Example (1/3)

Linux kernel 2.6.23 included the **Completely Fair Scheduler**:

- Uses a Red-Black tree as the data structure. Why?;
- Tree nodes are indexed by processor "execution time" in nanoseconds;
- When choosing a new process the scheduler:
  - Node with the lowest execution time (left-most) is chosen;
  - If the process completes execution it is removed from the system and tree;
  - If the process reaches its maximum execution time or is otherwise stopped:
    - it is reinserted into the scheduling tree based on its new spent execution time;
  - Otherwise, the new left-most node will then be selected from the tree.

Linux Scheduler Example (2/3)

If the process spends a lot of its time sleeping:

- Spent time value is low;
- Automatically gets the priority boost when it finally needs it.
- Hence such tasks do not get less processor time than the tasks that are constantly running.

Linux Scheduler Example (3/3)

CFS scheduler has a scheduling complexity of:

- $O(\log(N))$ where N is the number of processes;
- Choosing a task can be done in constant time, *i.e.* $O(1)$;
- Reinserting a task after it has run requires $O(\log(N))$ operations;
- These complexities are all a result of using Red-Black trees.

Threads

Each process has an address space and a single **thread** of control:

What if a process **blocks**? Why not run some other code of the process?

Threads

Each process has an address space and a single **thread** of control:

What if a process **blocks**? Why not run some other code of the process?

- Answer: **Threads**
- Threads can be thought of parallel entities within the process;

Threads

What if we wish to share the process data amongst different parallel entities?

Threads

What if we wish to share the process data amongst different parallel entities?

- Answer: **Threads**
- Threads share an address space and all of its data among themselves;
- Essential ability for certain applications;

Also: creating a process is a heavy/slow computational task:

- Allocate memory;
- Setup data / text memory sections;
- Setup file descriptors;
- Setup all the necessary resources;

Sometimes there is no need to copy the same data again:

What if we wish to save on all this time?

Also: creating a process is a heavy/slow computational task:

- Allocate memory;
- Setup data / text memory sections;
- Setup file descriptors;
- Setup all the necessary resources;

Sometimes there is no need to copy the same data again:

What if we wish to save on all this time?

- Answer: **Threads**
- Creating threads is 10 - 100 times faster;

What if we have multiple cores?

- With processes:
  - Each core could execute a process;
- With threads:
  - Each process can run multiple threads;
  - Each core could execute a thread;

Lets look at a specific example:

Can you give specific examples of threads for this game? Any ideas?

Can you give specific examples of threads for this game? Any ideas?

- One thread responsible for drawing visual elements;
- One thread for calculating physics;
- One thread for processing audio;
- One thread for processing keyboard inputs;
- One thread for multiplayer;
- Etc...

However: this strict partition model is not the best;

- Sometimes threads will not have anything to execute;
- As a result: threads will be idle;
- Idle threads are bad since the CPU is not being fully utilized;

Can you think of a better model? Any ideas?

Can you think of a better model? Any ideas?

Have a **dipatcher** thread that:

- Receives tasks that need to be performed;
- Dispatcher then:
  1. Chooses an idle **worker** thread:
  2. Wakes **sleeping** thread;
  3. When the worker wakes up:
     - Worker starts **executing** required task;
- As a result:
  - As soon as a task is received it is allocated to a specific thread;

Pseudo-code for **scheduler** and **worker** threads:

```
while(TRUE){
        get_next_request( &buffer );
        handoff_work( &buffer)

}


while(TRUE){
        wait_for_work( &buffer );
        execute_work( &buffer );
        return;
}
```

Classical Thread Model

From our previous slides we know that a **process** has an:

- **Address space** containing:
  - program text and data;
  - open files;
  - child processes;
  - alarms;
  - and more...
- As a result:
  - processes can be seen as a collection of related resources;

Processes also have a **thread of execution:**

- **Do not** confuse with having multiple threads;
- Single thread of execution has:
  - Program Counter;
  - Registers;
  - Function call stack;

In your opinion:

What are the differences between **processes** and **threads**? Any ideas?

What are the differences between **processes** and **threads**? Any ideas?

- **Processes** group resources together;
- **Threads** are the entities scheduled for CPU execution:
  - Allow multiple executions in the same process;
    - Instead of having a single thread of execution;
  - Because threads share the address space:
    - They have access to all of the processes' resources

Original model (i.e. single thread of execution):

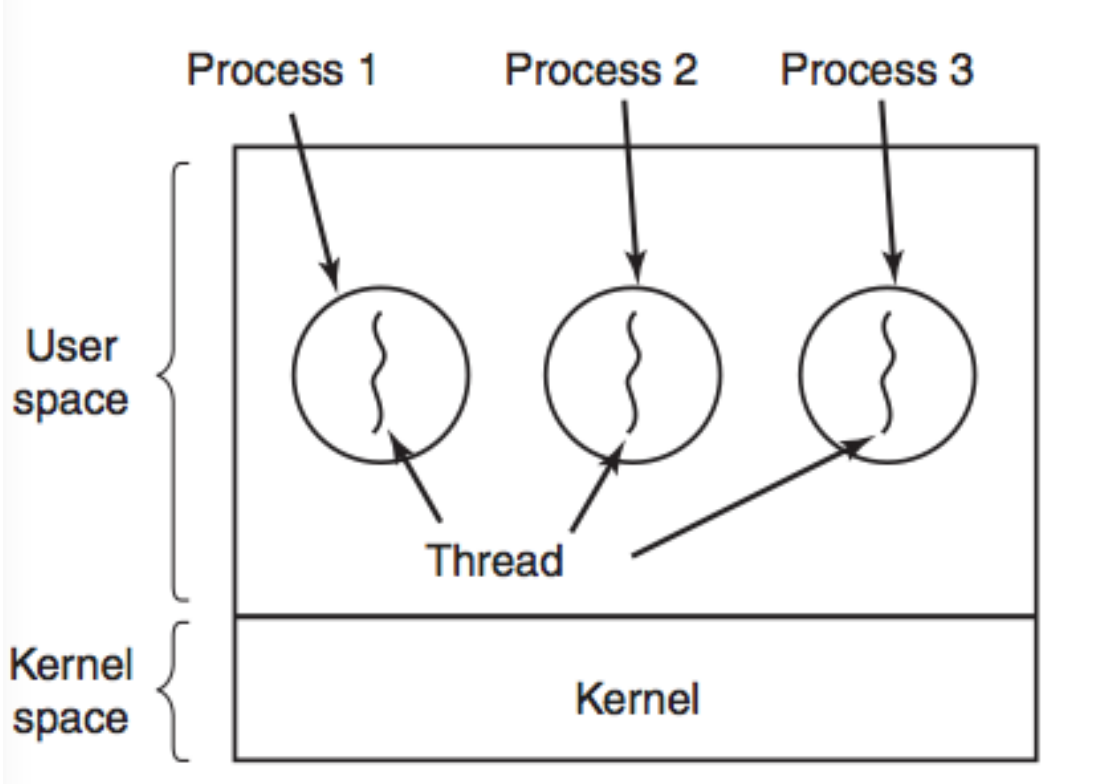

Figure: Three processes each with one thread. (Source: (Tanenbaum and Bos, 2015))

- Each process has its own address space;
- Each process has its own single thread of control;

Thread model (i.e. various threads of execution):

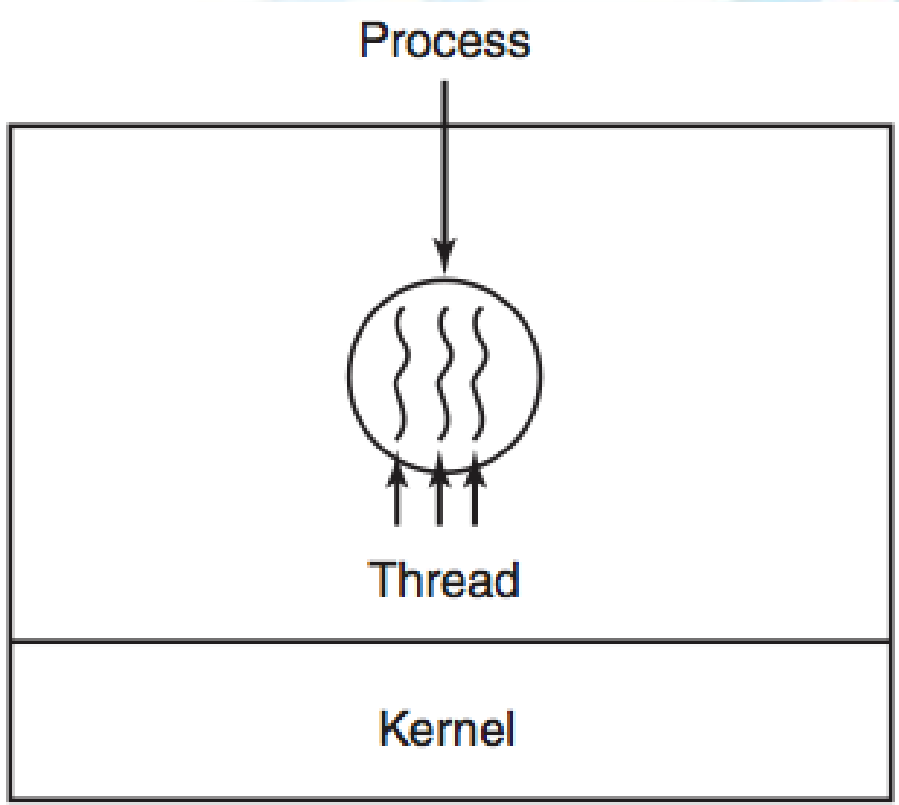

Figure: One process with three threads. (Source: (Tanenbaum and Bos, 2015))

- *E.g.:* a single process with three threads of control;
- All three threads operate in the same address space;

When a multithreaded process is run on a single-CPU system:

- threads take turns running;
  - similar to how having multiple processes work;
- gives the illusion of parallelism;

Because threads share the same address space:

- They also share the same global variables;
- This can lead to problems:

> Can you see what type of problems can occur? Any ideas?

Because threads share the same address space:

- They also share the same global variables;
- This can lead to problems:

> Can you see what type of problem can occur? Any ideas?

- One thread can read / write other thread's data;
- We will study this on later chapters =)

Each thread represents a different execution path:

What do you think the OS needs to keep track of? Any ideas?

Each thread represents a different execution path:

What do you think the OS needs to keep track of? Any ideas?

What information was needed for a process with a single thread of execution?

OS maintains per thread:

- Program counter;
- Registers;
- Function call stack;
- State;

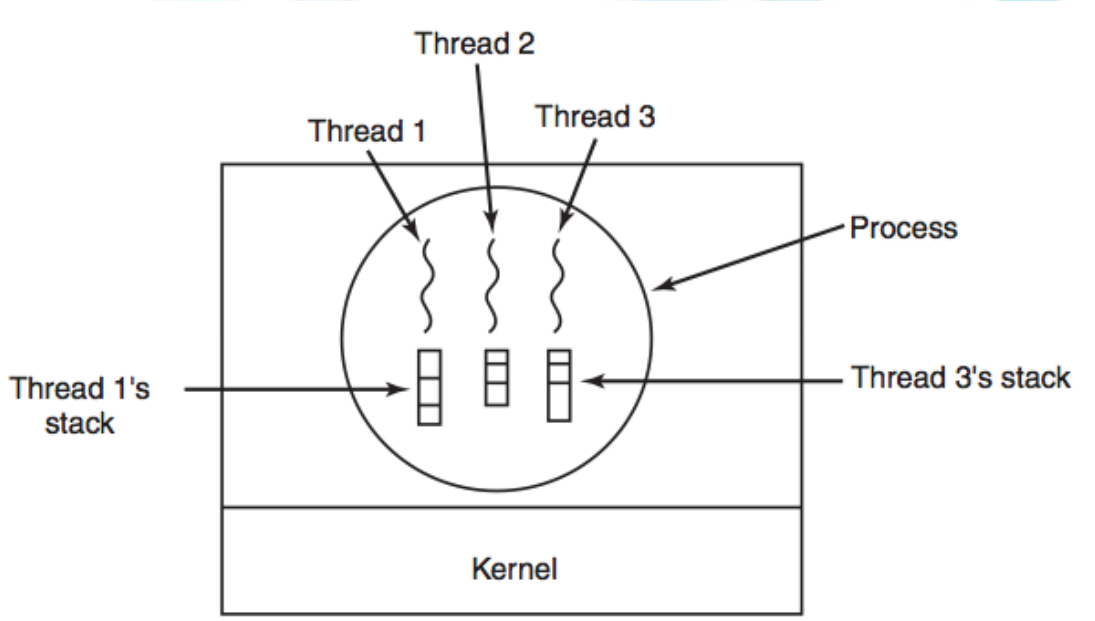

Figure: Each thread has its own stack. (Source: (Tanenbaum and Bos, 2015))

In **conclusion:**

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	

Figure: The first column lists some items shared by all threads in a process. The second one lists some items private to each thread.(Source: (Tanenbaum and Bos, 2015))

Do you see any other similarities between threads and processes?

Do you see any other similarities between threads and processes?

Like a traditional process, a thread can be in one of several states:

- running;
- blocked;
- ready;
- terminated;

Can you guess what each state represents?

- **Running:**
  - Thread is using the CPU;
- **Blocked:**
  - Thread is waiting for some event to unblock it;
- **Ready:**
  - Thread is scheduled to run but not yet running;
- **Terminated**

In essence the following procedures are required:

- **thread_create:**
  - Specifies the name of a procedure for the new thread to run;
- **thread_exit:**
  - Thread has finished its work and is no longer schedulable;
- **thread_join:**
  - Blocks the calling thread until a certain thread has exited;
- **thread_yield:**
  - Thread voluntarily gives up the CPU to let another thread run;

So the question now is:

How are threads implemented in Unix / Linux (posix) systems? Any ideas?

So the question now is:

How are threads implemented in Unix / Linux (posix) systems? Any ideas?

- The same way that we have POSIX systems calls...
- ...We also have POSIX threads...

Care to guess how the POSIX threads are named Any ideas?

So the question now is:

How are threads implemented in Unix / Linux (posix) systems? Any ideas?

- The same way that we have POSIX systems calls...
- ...We also have POSIX threads...

Care to guess how the POSIX threads are named Any ideas?

- **Pthreads** =)
- Guess what we will be seeing next ;)

POSIX threads

IEEE defined a standard for threads call **Pthreads**:

- IEEE is an international organization for defining standards;
- Portable Operating System Interface (**POSIX**):
  - Set of IEEE standards for maintaining compatibility between OS;
- POSIX threads or Pthreads:
  - Execution model that allows for a parallel execution model.
  - Available on many Unix-like POSIX-conformant OS:
    - *E.g.:* Linux, Mac OS X, Android and Solaris:
    - Typically bundled as library **libpthread**

Each Pthread thread has a set of attributes:

Can you guess some of these attributes? Any ideas?

Each Pthread thread has a set of attributes:

Can you guess some of these attributes? Any ideas?

- Set of registers (PC, IR, ...);
- Function call stack;
- Stack size;
- Thread identifier;
- Scheduling parameters, *e.g.:*
  - time thread has executed;
  - Priority;
- All these attributes are represented in a single entity:
  - Thread data type called **pthread_t**

Pthreads defines over 60 function calls:

- All of which you should know for your exam

Pthreads defines over 60 function calls:

- All of which you should know for your exam ;)

Here are the main ones:

Thread call	Description
Pthread_create	Create a new thread
Pthread_exit	Terminate the calling thread
Pthread_join	Wait for a specific thread to exit
Pthread_yield	Release the CPU to let another thread run
Pthread_attr_init	Create and initialize a thread's attribute structure
Pthread_attr_destroy	Remove a thread's attribute structure

Figure: Some of the Pthreads function calls (Source: (Tanenbaum and Bos, 2015))

Guess what we will be seeing next? ;)

pthread_create:

- Creates a new thread;
- Thread identifier of the new thread is returned;
- Extremely Basic Example:

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

pthread_t thread;

pthread_create( &thread, NULL, NULL, NULL )
```

Can you see anything ``wrong'' with this example?

Can you see anything ``wrong'' with this example?

- Threads are supposed to perform some computation...
- We need to tell threads what function to compute:
  - This can be done by pointing thread to a function;
  - **Important:**
    - Function must return **void***
    - Function must take a single argument of type **void***
- We can refine the previous example;

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

void* isPrimeNumber( void* argument ){ ... }

pthread_t thread;

/* Check whether the first 1000 numbers are prime */
for( int counter = 0; counter < 1000; counter++ ){

        pthread_create( &thread, NULL, isPrimeNumber, &counter )

}
```

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

void* isPrimeNumber( void* argument ){ ... }

pthread_t thread;

/* Check whether the first 1000 numbers are prime */
for( int counter = 0; counter < 1000; counter++ ){

        pthread_create( &thread, NULL, isPrimeNumber, &counter )

}
```

Can you see anything wrong with this code? Any ideas?

Can you see anything wrong with this code? Any ideas?

- System calls may fail;
- Important to test if function has failed or not;
  - If **successful**
    - **pthread_create()** function will return zero
  - **Otherwise:**
    - An error number will be returned to indicate the error.
- We can refine the previous example;

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

void* isPrimeNumber( void* argument ){ ... }

pthread_t thread;

/* Check whether the first 1000 numbers are prime */
for( int counter = 0; counter < 1000; counter++ ){

        if( pthread_create( &thread, NULL, isPrimeNumber, &counter ) != 0 ){

                printf("Error creating thread\n");

                }
        }
}
```

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

void* isPrimeNumber( void* argument ){ ... }

pthread_t threads( 1000 );

/* Check whether the first 1000 numbers are prime */
for( int counter = 0; counter < 1000; counter++ ){

        if( pthread_create( &threads( counter ), NULL, isPrimeNumber, &counter ) != 0 ){

                printf("Error creating thread\n");

                }
        }
}
```

Can you see anything wrong with this example? Any ideas?

Can you see anything wrong with this example? Any ideas?

- Threads are created;
- Threads should eventually stop;
- Previous code:
  - Just created threads;
  - Does not wait for threads to terminate;
  - This is done through two function calls:
    - **pthread_exit**: terminates a thread;
    - **pthread_join**: waits for a thread to terminate;
- We can thus refine our previous code;

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

void* isPrimeNumber( void* argument ){

        int result = calculateIfArgumentIsPrimeNumber( (int) *argument )

        pthread_exit( &result );
}

pthread_t threads( 1000 );
int results( 1000 )

/* Check whether the first 1000 numbers are prime */
for( int counter = 0; counter < 1000; counter++ ){

        if( pthread_create( &threads( counter ), NULL, isPrimeNumber, &counter ) != 0 ){

                printf("Error creating thread\n");

        }
}

/* wait for all threads to finish */
for( int counter = 0; counter < 1000; counter++ ){

        if( pthread_join( threads( counter ), (void**)&( results( counter ) ) ) ) {

                printf("Error joining thread\n");
                return ERROR;

        }else{ printf(``Result:%d\n'', results( counter ))}
}
```

Other system calls:

- **pthread_yield**: give another thread a chance to run;
- **pthread_attr_init**: creates and initializes attribute structure of a thread;
- **pthread_attr_destroy**: frees memory from attribute structure of a thread;

Lets see if you understood all of these concepts:

Can you tell me what the following code is doing? Any ideas?

Can you tell me what the following code is doing? Any ideas?

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS 10

void* print_hello_world(void*tid) {
        printf("Hello World. Greetings from thread %d\n", tid); pthread exit(NULL);
}

int main(int argc, char *argv()) {

pthread_t threads( NUMBER OF THREADS );
int status, i;

for( i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads( i ), NULL, print_hello_world, (void *) i );

        if (status != 0) {
                printf("Oops... pthread create returned error code %d\n", status);
                exit(-1);
        }
}
exit(NULL);
}
```

Previous code:

- Creates NUMBER_OF_THREADS pthreads;
- Initializes them to run print_hello_world function;
- If the thread creation fails: prints an error message and exits;

Important: When compiling pthread programs:

- You will need to add library **libpthread**;
- This can be done via the compile command:

```
$ gcc program.c -o program -lpthread
```

Lets talk about additional concepts:

- There are two main methods to implement threads:
  - Threads in User Space;
  - Threads in Kernel Mode;

Guess what we will be seeing next? ;)

Threads in User Space

Threads package exist entirely in user space:

- Kernel:
  - Knows nothing about threads;
  - Is managing ordinary single-threaded processes;

Can you see any advantage of using this method?

Threads in User Space

Threads package exist entirely in user space:

- Kernel:
  - Knows nothing about threads;
  - Is managing ordinary single-threaded proceses;

> Can you see any advantage of using this method?

- Threads can be implemented in an OS that does not support threads;
- All OS **used** to fall in this category;

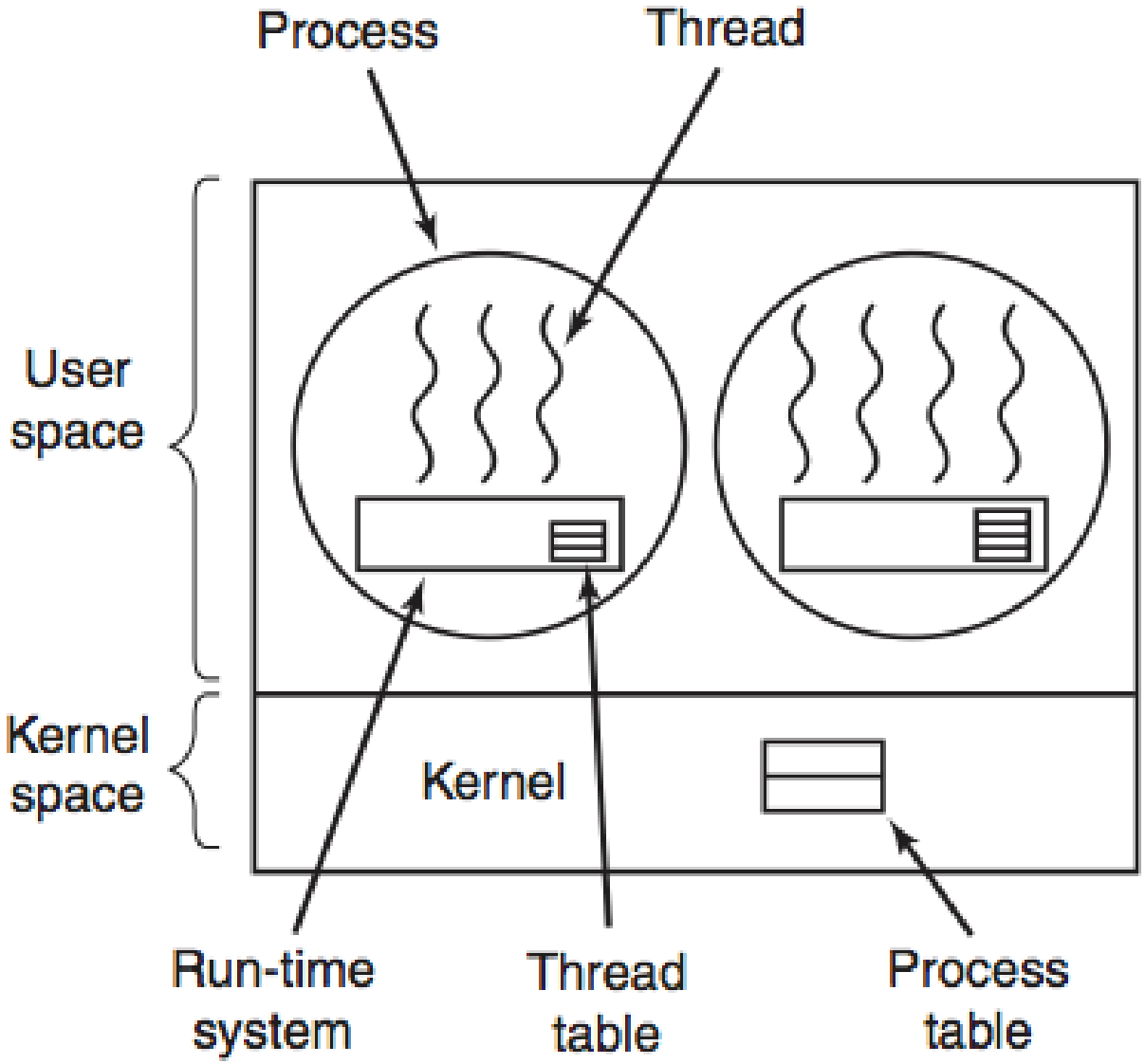

Figure: User-level thread package (Source: (Tanenbaum and Bos, 2015))

From the previous picture (1/2):

- Threads run on top of OS;
- Threads are implemented by a library:
  - thread_create;
  - thread_join;
  - etc;
- Each process needs its own thread table, keeping track of each thread:
  - PC, SP, IR;
  - state (blocked, running, finished, etc) ;
  - etc;

From the previous picture (2/2):

- Thread table is managed by run-time system;
- Threads are switched based on their state, execution time and others;
- If machine has instructions to save / load all registers:
  - Only thread context is switched (little information);
  - No need to switch context to kernel (much bigger information):
    - **Result:** much faster than trapping to the kernel;

Other advantages:

- Each process can have its own scheduler;
- Scale better since storing all information in kernel may be problematic;

Can you see any disadvantages? Any ideas?

Can you see any disadvantages? Any ideas?

- Blocking system calls will block threads:
  - Precisely what we were trying to avoid by using calls;
  - Cannot be changed without changes to OS;
- Page faults:
  - Kernel blocks entire process to fetch information;
  - But other threads belonging to the same process could be executed;

Threads in the Kernel

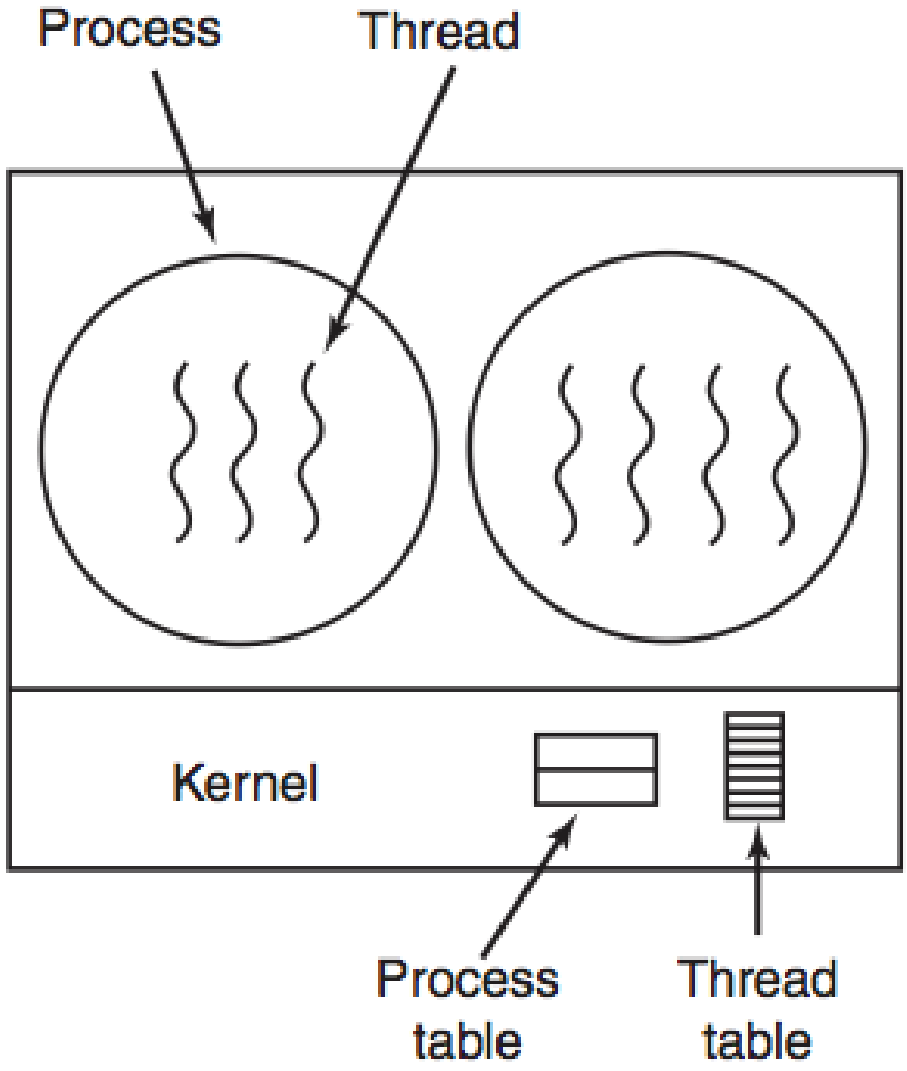

Figure: Threads packaged managed by the kernel (Source: (Tanenbaum and Bos, 2015))

From the previous picture (1/2):

- Kernel has a thread table;
- Thread management is done through a kernel call:
  - Much slower: since kernel context needs to be loaded;
- Kernel can detect blocking system calls and switch threads:
  - Switching to threads of the same process;
  - Or switching to threads belonging to other processes;

From the previous picture (2/2):

- With user-level threads:
  - Process threads keep running until kernel takes CPU away;
- If one thread in a process causes a page fault:
  - Kernel can choose another runnable thread;

Hybrid Implementations

Idea: use kernel-level threads and multiplex user-level threads in them:

- Try to get only advantages and mitigate disadvantages;
- Programmer determines:
  - How many kernel threads to use;
  - How many user-level threads to use;
- Kernel:
  - Aware of kernel-level threads and schedules those;
  - Some of those threads may have multiple user-level threads:

Making Single Threaded Code Multithreaded

Many existing programs were written for single-threaded processes:

- Converting to multithreading is very tricky

Thread code normally consists of multiple procedures with:

- Local variables, global variables, and parameters;
- Global variables are problem:
  - Lets see why.

Example

Consider the **errno** variable maintained by UNIX:

- When a system call fails: error code is put into errno.
- Now lets consider the following scenario:

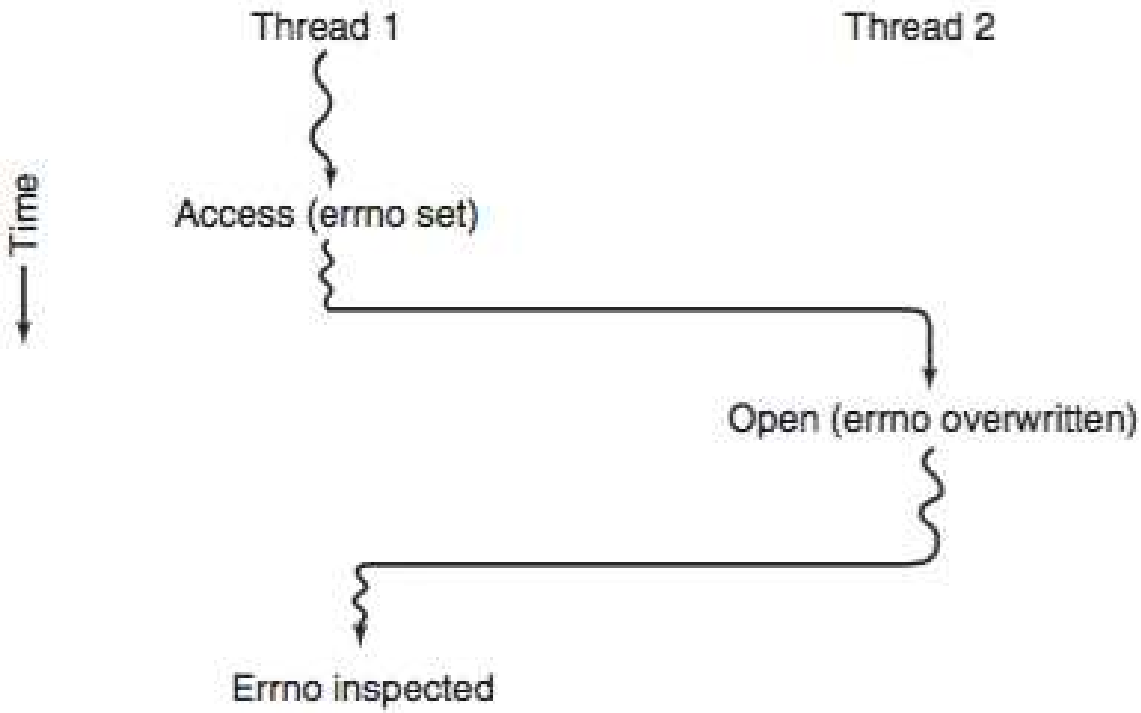

Figure: Conflicts between threads over the use of a global variable. (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- Thread 1 executes system call **access** to check file permissions;
- OS returns answer in global variable **errno**;
- After control has returned to thread 1, but before **errno** is read:
  - Scheduler switches to thread 2;
  - Thread 2 executes an **open** system call that fails:
    - This causes **errno** to be overwritten;
    - Thread 1 access code is lost!
    - When thread 1 resumes it will read wrong value and behave incorrectly!

How can we solve this problem? Any ideas?

How can we solve this problem? Any ideas?

- One easy solution: **prohibit** global variables:
  - Possible conflicts with existing software;
  - Not a very good solution. What else can be done?

How can we solve this problem? Any ideas?

- One easy solution: **prohibit** global variables:
  - Possible conflicts with existing software;
  - Not a very good solution. What else can be done?
- Another solution: each thread has its own private global variables
  - Lets have a look at this approach;

Another solution: each thread has its own private global variables

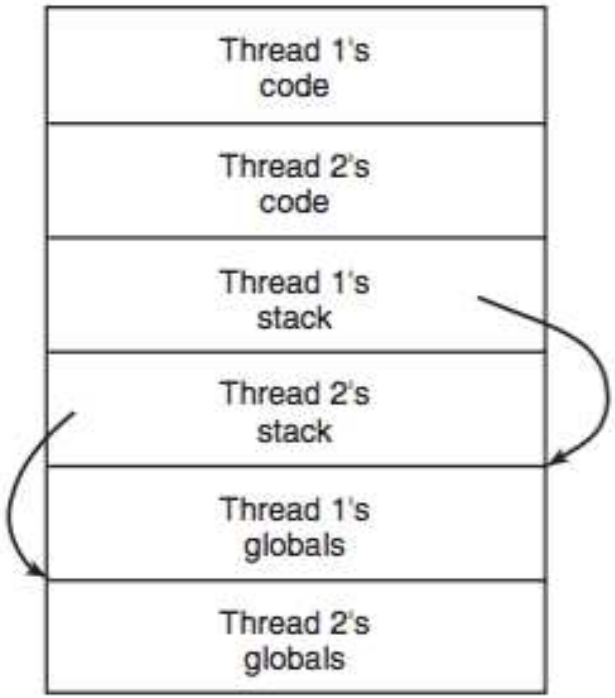

Figure: Threads can have private global variables. (Source: (Tanenbaum and Bos, 2015))

- Each thread has its own **errno** variable and other global variables;

Another solution: procedures to create, set and read global variables:

```
create_global(``variable'')
set_global(``variable'', &buffer )
buffer = read_global(``variable'')
```

- Only the calling thread has access to the global variable;
- If another thread creates a global variable with the same name:
  - Variable is mapped to a different memory position

Can you see any other problems besides global variables? Any ideas?

Can you see any other problems besides global variables? Any ideas?

- Many library procedures are not **reentrant**:
  - I.e. not designed for additional calls before the original call has finished,
  - Library variables such as buffers will be reused between different threads!
    - Threads will rewrite these variables, which is a big problem;
  - Nontrivial activity to rewrite all libraries:
    - Bugs may be introduced...

What can we do besides rewriting entire libraries? Any ideas?

What can we do besides rewriting entire libraries? Any ideas?

- Whenever a thread uses a library we can set a bit to one;
- Any attempt by another thread to use the library:
  - Is blocked until library is freed;
- However: greatly eliminates potential parallelism... =(

What about signals? How should signals be adapted from single to multi-threading?

What about signals? How should signals be adapted from single to multi-threading?

- Who should catch the signals generated from timers, keyboard interrupts and others?
  - One designated thread?
  - All the threads?
  - The latest created thread?
- Managing signals is difficult enough in a single-threaded environment:
  - Going to a multithreaded environment only makes this worse...

Also, what are the implications with stack management? Any ideas?

Also, what are the implications with stack management? Any ideas?

Usually: when a process' stack overflows

- Kernel just provides that process with more stack automatically;
- When a process has multiple threads, it must also have multiple stacks;
- If the kernel is not aware of all these stacks:
  - Cannot grow them automatically upon stack fault;
  - May not even realize that a memory fault is related to the growth of some thread's stack.

Conclusion:

- These problems are not insurmountable...
- However, these problems show that:
  - Difficult to introduce threads to a single-threaded environment...
  - ...Without substantial system redesign;

Now that we have talked a little about threads:

- Lets look at another topic:
  - Interprocess communication;

Interprocess Communication

As we previously saw:

- Processes may need to communicate with other processes:

Essentially, there are three issues here (1/3):

> How to pass information from one process to another?

Interprocess Communication

As we previously saw:

- Processes may need to communicate with other processes:

Essentially, there are three issues here (2/3):

> How to guarantee that two processes do not interfere with each other?

Interprocess Communication

As we previously saw:

- Processes may need to communicate with other processes:

Essentially, there are three issues here (3/3):

> How to guarantee proper sequencing when dependencies are present?

Interprocess Communication

As we previously saw:

- Processes may need to communicate with other processes:

Essentially, there are three issues here:

- How to pass information from one process to another?
- How to guarantee that two processes do not interfere with each other?
- How to guarantee proper sequencing when dependencies are present?

These issues are known as **InterProcess Communication (IPC)**

- The same problems and solutions also apply to threads;

Race Conditions

Lets see how IPC works with a printer spooler (1/5):

- Program that feeds files for a printer to print;
- When a process wants to print a file:
  - Process enters file name in a directory;
- Another process, the **printer daemon**:
  - Periodically checks if there are files to be printed:
    - If there are: files are printed and names removed from directory;

Race Conditions

Lets see how IPC works with a printer spooler (2/5):

- Directory has a very large number of slots:
  - Numbered 0, 1, 2, ... with each capable of holding a file name;
- There are two shared variables:
  - **out**: points to the next file to be printed;
  - **in**: points to the next free directory slot;

Race Conditions

Lets see how IPC works with a printer spooler (3/5):

- At a certain instant:
  - Slots 0 to 3 are empty:
    - the files were printed
  - Slots 4 to 6 are full
    - with the names of files queued for printing
  - More or less simultaneously:
    - Processes A and B decide they want to queue a file for printing;

Race Conditions

Lets see how IPC works with a printer spooler (4/5):

- More or less simultaneously:
  - Processes A and B decide they want to queue a file for printing;

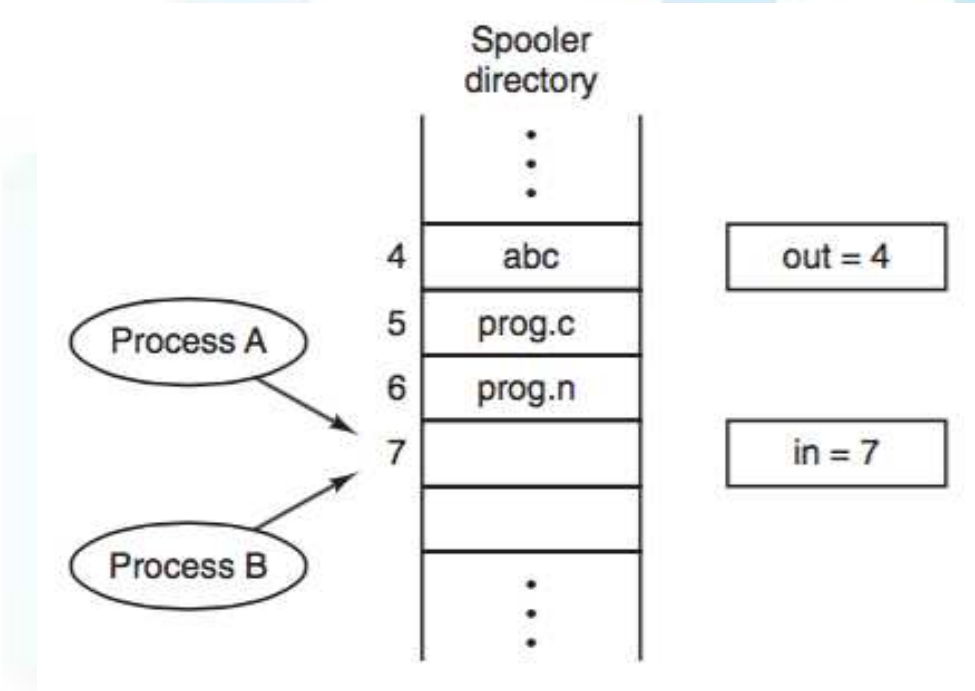

Figure: Two processes want to access shared memory at the same time. (Source: (Tanenbaum and Bos, 2015))

Lets see how IPC works with a printer spooler (5/5):

- The following could happen:
  - Process A reads **in** and sees the value 7;
  - OS switches to process B;
  - Process B reads **in** and also sees the value 7;
  - At this instant both processes think the next available slot is 7;
  - Process B continues to run:
    - Stores file name in slot 7 and updates **in** to 8;
  - Eventually: process A runs again:
    - Stores file name in slot 7 and updates **in** to 8;
  - **Conclusion:** Process B's file will never get printed;

Situations like this:

- Where two or more processes are reading / writing some shared data

Are called **race conditions**:

- Debugging programs with race conditions is a **nightmare**;
- Everything will seem alright:
  - But eventually something weird will happen...
- With increasing parallelism due to increasing number of cores:
  - Race conditions are becoming more common...

How do we avoid race conditions? Any ideas?

How do we avoid race conditions? Any ideas?

- **Key:** whenever we have **shared data:**
  - Prohibit reading / writing at the same time;
  - This is known as **mutual exclusion**
- Choice of appropriate operations for achieving mutual exclusion:
  - Represents a major design issue in an OS!

In abstract terms, part of the time:

- Process is busy doing things that do not produce race conditions;
- However, sometimes a process has to access shared data;
- The part of the program where shared data is accessed is called:
  - **critical region**

Now I can ask again the same question:

> How do we avoid race conditions? Any ideas?

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions.

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions.
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.

We need four conditions to hold to have a good solution:

- No two processes may be simultaneously inside their critical regions:
  - Accessing shared data implies processes share critical regions;
- No assumptions may be made about speeds or the number of CPUs.
- No process running outside its critical region may block any process.
- No process should have to wait forever to enter its critical region.

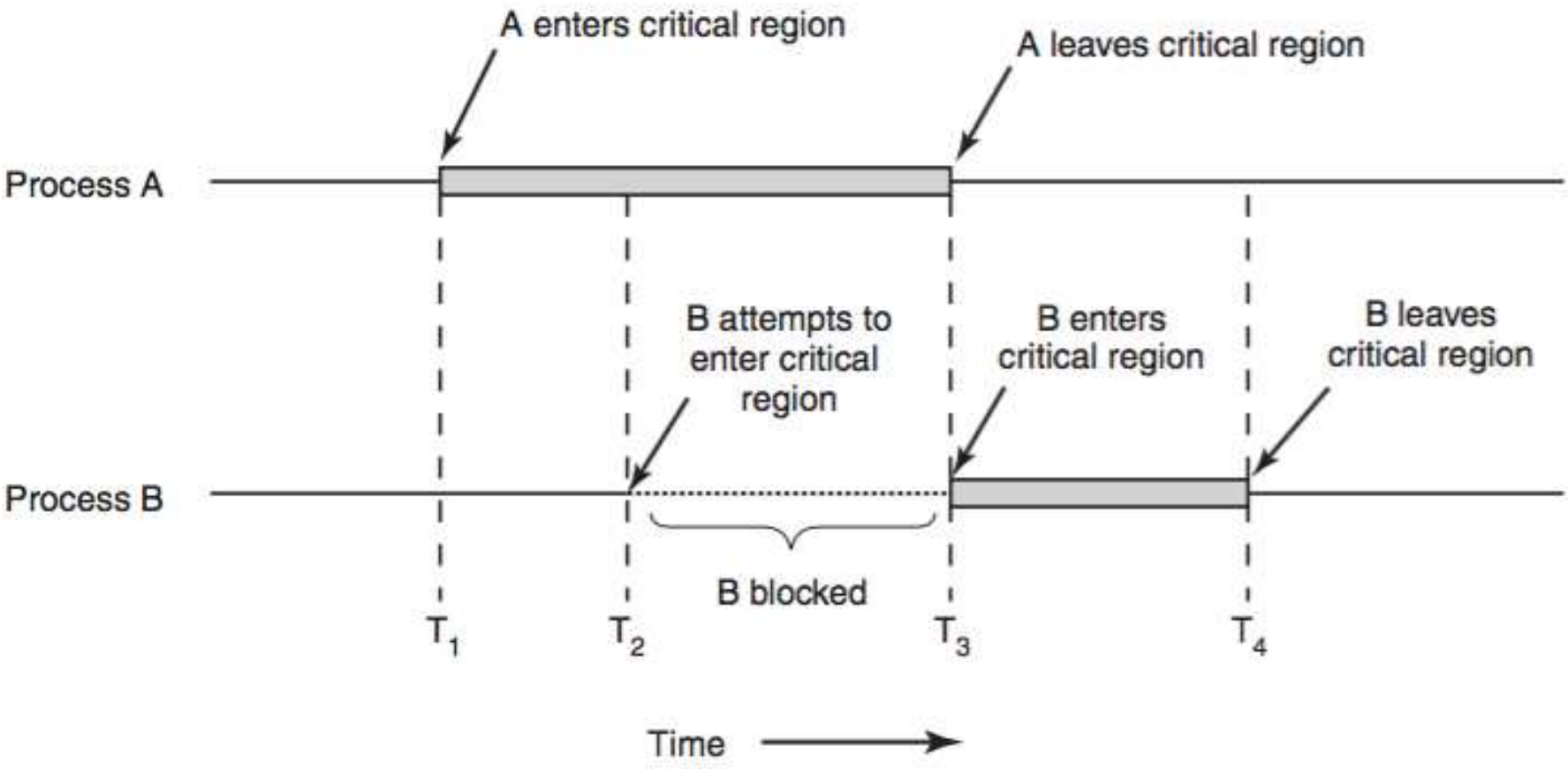

Figure: Mutual exclusion using critical regions (Source: (Tanenbaum and Bos, 2015))

From the previous figure:

- Process A enters critical region at T_1;
- Process B tries to enter critical region at T_2:
  - But **fails** since A is already in the critical region;
  - Consequently: B is temporarily suspended until T_3:
    - Time when A leaves its critical region, allowing B to enter immediately;
  - Eventually B leaves (at T_4) and no process is in a critical region;

So the question now is:

How do we implement mutual exclusion? Any ideas?

Mutual Exclusion with Busy Waiting

There are several mutual exclusion mechanisms guaranteeing that:

- While one process is busy updating shared memory in its critical region:
  - No other process will enters its critical region and cause trouble;
- Lets have a look at the following:
  - Disabling interrupts;
  - Lock variables;
  - Strict alternation;
  - TSL instruction;

Disabling interrupts

Have each process disable interrupts before entering critical region:

- Re-enable interruptions before leaving critical region;
- With interrupts disabled no clock interrupts can occur;
- CPU is only switched from process to process as a result of a clock;
- If interrupts are turned off, the CPU will not switch to another process;
- **Conclusion:** No other process will enter critical region

Can you see any problem with disabling interrupts? Any ideas?

Can you see any problem with disabling interrupts? Any ideas?

- Unwise to give user processes power to turn off interrupts:
  - What if the interrupts are never restored?
  - What if the process crashes?
- If the system is a multiprocessor:
  - Disabling interrupts affects only CPU with the **disable** instruction;
  - Other one will continue running and can access shared memory;

Disabling interruptions should only be done by the kernel:

- To update internal variables;
- *E.g.:* Disable interrupts to update list of ready processes:
  - This way no inconsistent state exists;

Lock Variables

Idea: have a single shared variable called **lock**:

- Initialized to zero;
- When a process wants to enter its critical regions:
  - Tests the lock;
  - If the lock is 0:
    - process sets it to 1 and enters critical regions;
  - If the lock is 1:
    - process waits until it becomes 0;

Can you see any problem with the lock variables strategy? Any ideas?

Can you see any problem with the lock variables strategy? Any ideas?

- Exact same problem with the spooler directory;
- Lock variable is a shared data:
  - Susceptible to race conditions!

Strict Alternation

```
while (TRUE) {
    while (turn != 0)        /* loop */ ;
    critical_region();
    turn = 1;
    noncritical_region();
}
```

(a)

```
while (TRUE) {
    while (turn != 1)        /* loop */ ;
    critical_region();
    turn = 0;
    noncritical_region();
}
```

(b)

Figure: A proposed solution to the critical-region problem (Source: (Tanenbaum and Bos, 2015))

- Variable **turn**, initially zero:
  - Keeps track of whose turn it is to enter the critical region;
- Process 0 inspects **turn**, finds it to be zero and enters its critical region;
- Process 1 also finds it to be zero and continually tests **turn** until it becomes 1;
- Continuously testing a variable is called **busy waiting**;

Can you see any problems with the strict alternation approach? Any ideas?

Can you see any problems with the strict alternation approach? Any ideas?

- Loops are a waste of CPU time!

Can you see any problems with the strict alternation approach? Any ideas?

- Loops are a waste of CPU time!

Can you see any **other** problems with the strict alternation approach? Any ideas?

Can you see any **other** problems with the strict alternation approach? Any ideas?

- Assume both processes are in their noncritical regions with **turn** set to 0;
- Assume process 0 executes its whole loop quickly and sets **turn** to 1:
  - At this point **turn** is 1 and both processes are in noncritical regions;
- If process 0 continues it is not permitted to enter its critical region:
  - Process 0 is being blocked by a process not in its critical region;
  - **Conclusion:** Violates condition 3!!!

TSL Instruction

Some computers have an instruction like:

```
TSL RX, LOCK
```

- **Test and Set Lock** works as follows:
  - Reads contents of the memory word lock into register RX;
  - Stores a nonzero value in lock address;
- Reading and storing operations are guaranteed to be indivisible:
  - No other processor can access the memory word until instruction is finished;
  - CPU executing TSL locks memory bus:
    - Prohibiting other CPUs from accessing memory;

TSL instruction is used alongside a shared variable **lock**:

- To coordinate memory access to shared memory;
- When **lock** is 0:
  - Any process may set it 1 using TSL instruction;
  - And then read / write the shared memory;
- When process is done it sets **lock** back to zero;

How can this method be used to guarantee mutual exclusion? Any ideas?

```
enter_region:
        TSL REGISTER,LOCK       | copy lock to register and set lock to 1
        CMP REGISTER,#0         | was lock zero?
        JNE enter_region        | if it was not zero, lock was set, so loop
        RET                     | return to caller; critical region entered

leave_region:
        MOVE LOCK,#0            | store a 0 in lock
        RET                     | return to caller
```

Figure: Entering and leaving a critical region using the TSL instruction.(Source: (Tanenbaum and Bos, 2015))

From the previous figure (1/2):

1. 1^{st} instruction copies old lock value and then sets lock to 1:
   - Old value is compared with 0;
   - If lock is nonzero:
     - lock was already set and program waits until lock is free;
     - Sooner or later lock will become 0 **(busy waiting)**;
   - Otherwise:
     - Lock is set and process / thread enters critical region;
     - When process leaves critical region: Lock is set to 0;

From the previous figure (2/2):

2. Overall solution:
   - Processes / Threads must call at the correct times:
     - **enter_region** before entering critical region;
     - **leave_region** before leaving critical region;

From the previous figure (2/2):

2. Overall solution:
   - Processes / Threads must call at the correct times:
     - **enter_region** before entering critical region;
     - **leave_region** before leaving critical region;

But what happens if one process cheats? Any ideas?

From the previous figure (2/2):

2. Overall solution:
   - Processes / Threads must call at the correct times:
     - enter_region before entering critical region;
     - leave_region before leaving critical region;

> But what happens if one process cheats? Any ideas?

   - Mutual exclusion will **fail**: processes must cooperate!

Sleep and Wakeup

Can you any problems with the previous approach? Any ideas?

Sleep and Wakeup

Can you any problems with the previous approach? Any ideas?

Busy waiting:

- When a process wants to enter critical region:
  - Checks to see if the entry is allowed:
    - If it is not, the process just sits in a tight loop **waiting**;
    - Wasteful of processor time!

What can we do to circumvent the busy waiting approach? Any ideas?

What can we do to circumvent the busy waiting approach? Any ideas?

- **Block** processes instead of having them on a busy wait;
- This is done through **sleep** and **wakeup** OS primitives;

Sleep:

- System call that causes the caller to block;
- Process is suspended until another process wakes it up;
- Scheduler can choose another process that is in ready state;

Wakeup:

- Wakes another process;
  - Has one parameter: process to be awakened.

Producer-consumer problem

These primitives can be exemplified through the **producer-consumer** problem:

- Two processes share a common buffer;
  - **Producer:** puts information into the buffer;
  - **Consumer:** takes information out of the buffer;

What happens when **producer** wants to put a new item in the buffer, but it is already **full**?

What happens? A **problem** happens!

What can we do to solve the problem? Any ideas?

What happens when **producer** wants to put a new item in the buffer, but it is already **full**?

What happens? A **problem** happens!

What can we do to solve the problem? Any ideas?

The solution is for the producer to go to **sleep:**

- and **awakened** when **consumer** has removed one or more items;

What happens when the **consumer** wants to take an item of the buffer, but it is **empty**?

What happens? A **problem** happens!

What can we do to solve the problem? Any ideas?

What happens when the **consumer** wants to take an item of the buffer, but it is **empty**?

What happens? A **problem** happens!

What can we do to solve the problem? Any ideas?

The solution is for the consumer to go to **sleep:**

- and **awakened** when **producer** has put something into the buffer!

So now the question is:

How do we solve the consumer-producer problem? Any ideas?

First: lets define the following:

- **count** variable keeps track of the number of items in the buffer:
- **N** is the maximum number of items the buffer can hold;

Producer's code will first test to see if count is N:

- If it is: **producer** will go to **sleep**;
- If it is not: **producer** will add an item and increment count.

Consumer's code will first test to see if count is 0:

- If it is: **consumer** will go to **sleep**;
- If it is not: **consumer** will consume an item and decrement count.

Each process tests to see if the other should be awakened:

- and if so, wakes it up.

```
#define N 100                                   /* number of slots in the buffer */
int count = 0;                                  /* number of items in the buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          item = produce_item();                /* generate next item */
          if (count == N) sleep();              /* if buffer is full, go to sleep */
          insert_item(item);                    /* put item in buffer */
          count = count + 1;                    /* increment count of items in buffer */
          if (count == 1) wakeup(consumer);     /* was buffer empty? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repeat forever */
          if (count == 0) sleep();              /* if buffer is empty, got to sleep */
          item = remove_item();                 /* take item out of buffer */
          count = count - 1;                    /* decrement count of items in buffer */
          if (count == N - 1) wakeup(producer); /* was buffer full? */
          consume_item(item);                   /* print item */
     }
}
```

Figure: Producer-consumer problem with a fatal race condition.(Source: (Tanenbaum and Bos, 2015))

Can you see any problems with the previous code? Any ideas?

The following situation could possibly occur (1/2):

- Buffer is empty and the **consumer** has just read count to see if it is 0:
- At that instant, the scheduler decides to stop running the **consumer**:
  - And starts running the producer:
    - Producer inserts an item in the buffer...
    - ...increments count, and notices that it is now 1...
    - ...reasoning that count was just 0: producer wakes consumer...
    - ...consumer is not yet logically asleep, so the wakeup signal is lost.

The following situation could possibly occur (2/2):

- When **consumer** next runs:
  - it will test the value of count it previously read:
    - find it to be 0, and go to sleep
- Sooner or later the producer will fill up the buffer and also go to sleep:
  - **Both will sleep forever.**

Do you have any basic idea of how to solve the previous problem? Any ideas?

Do you have any basic idea of how to solve the previous problem? Any ideas?

- Everything would be fine if the wake signal had not been lost...

Semaphores

Semaphores data type were introduced to solve the previous problem:

- A semaphore could have the value 0:
  - indicating that no wakeups were saved
- Or some positive value if one or more wakeups were pending;
- Semaphores have two possible operations: **Down** and **Up**;

Down operation (1/2):

- If the value is greater than 0:
  - Value is decremented (one wakeup is used) and just continues.
- If the value is 0:
  - Process is put to sleep without completing the down for the moment;

Down operation (2/2):

- All semaphore operations are done as a **single indivisible atomic action**:
  - Checking the value;
  - Changing the value;
  - Possibly going to sleep;
- No other process can access the semaphore until the operation has completed or blocked
- OS guarantees this by implementing operation as a system call:
  - Control is not on the user-level side...

Atomic actions

- Group of related operations where:
  - All operations performed are done without interruption;
  - or not performed at all;
- Fundamental concept in many areas of computer science;

Up operation (1/2):

- Increments value of the semaphore;
- If one or more processes were sleeping on that semaphore:
  - One is chosen by the system and allowed to complete its down;
  - I.e.: after an up on a semaphore with processes sleeping on it:
    - Semaphore will still be 0;
    - But there will be one fewer process sleeping on it;
- Incrementing the semaphore and waking up one process is also indivisible:

How are the up / down operations performed in an indivisible way?

How are the up / down operations performed in an indivisible way?

OS guarantees this by implementing operation as a system call (1/3):

- Control is not on the user-level side...
- OS disables interrupts while it is:
  - Testing the semaphore;
  - Updating the semaphore;
  - Putting the process to sleep;

How are the up / down operations performed in an indivisible way?

OS guarantees this by implementing operation as a system call (2/3):

- As all of these actions take only a few instructions:
  - No harm is done in disabling interrupts.

How are the up / down operations performed in an indivisible way?

OS guarantees this by implementing operation as a system call (3/3):

- If multiple CPUs are being used:
  - Disabling the interrupts will only work for one CPU;
  - Therefore access to the system bus should forbidden:
    - Protect each semaphore by a lock variable with the TSL instruction;
    - Semaphore operation will take only a few microseconds of busy waiting;

Now the question is:

> How can we solve the producer-consumer problem using semaphores? Any ideas?

We need to identify:

- When to put producer / consumer to **sleep**;
- When to **awake** producer / consumer;

Now the question is:

> How can we solve the producer-consumer problem using semaphores? Any ideas?

Maybe we need to (1/2):

- Put **producer** to **sleep** when no more empty slots exist:
  - One semaphore to represent number of **empty slots**;
- Put **consumer** to **sleep** when no more full slots exist:
  - One semaphore to represent number of **full slots**;

Now the question is:

> How can we solve the producer-consumer problem using semaphores? Any ideas?

Maybe we need to (2/2):

- **Awake producer** when one item was consumed:
  - One semaphore to represent number of **empty slots**;
- **Awake consumer** when one item was produced:
  - One semaphore to represent number of **full slots**;

Are these two semaphores enough? Or is there still something missing? Any ideas?

Are these two semaphores enough? Or is there still something missing? Any ideas?

- Up / Down operations are done in atomic way:
  - So this is not what it is missing...

Are these two semaphores enough? Or is there still something missing? Any ideas?

- Up / Down operations are done in atomic way:
  - So this is not what it is missing...

But what about when we access the data structure representing the buffer?

Are these two semaphores enough? Or is there still something missing? Any ideas?

- Up / Down operations are done in atomic way:
  - So this is not what it is missing...

But what about when we access the data structure representing the buffer?

- Data buffer is a **critical region**;
- Needs to be accessed in **mutual exclusion**;
- This can be done through a binary semaphore, a.k.a. **mutex**;

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE is the constant 1 */
        item = produce_item();                  /* generate something to put in buffer */
        down(&empty);                           /* decrement empty count */
        down(&mutex);                           /* enter critical region */
        insert_item(item);                      /* put new item in buffer */
        up(&mutex);                             /* leave critical region */
        up(&full);                              /* increment count of full slots */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* infinite loop */
        down(&full);                            /* decrement full count */
        down(&mutex);                           /* enter critical region */
        item = remove_item();                   /* take item from buffer */
        up(&mutex);                             /* leave critical region */
        up(&empty);                             /* increment count of empty slots */
        consume_item(item);                     /* do something with the item */
    }
}
```

This solution uses three semaphores:

- **Full** for counting the number of slots that are full;
  - Initialized to zero;
- **Empty** for counting the number of slots that are empty;
  - Initialized to N;
- **Mutex** ensures producer/consumer do not access the buffer simultaneously;
  - Initialized to 1;
  - If each process:
    - does a down just before entering its critical region and...
    - ...an up just after leaving it...
    - **mutual exclusion** is guaranteed.

Mutex semaphore is used for **mutual exclusion**:

- Only one process at a time will be reading or writing the buffer;

Other **semaphores** are for **synchronization**:

- **full** and **empty** semaphores are needed to:
  - Guarantee that certain event sequences do or do not occur.
  - Ensure **producer** stops running when the buffer is full;
  - Ensure **consumer** stops running when it is empty;

POSIX systems have semaphores:

- Library: semaphore.h
- Data type: sem_t
- Methods:
  - sem_init
  - sem_wait
  - sem_post
  - sem_getvalue
  - sem_destroy

Mutexes

But what if we don't need the semaphore's ability to count?

What if we only need to manage mutual exclusion for some shared resource?

Mutex

Mutex is a shared variable that can be in one of two states:

- **Unlocked**
- **Locked**

Thread / process calls **mutex_lock** to access a critical region:

- If mutex is **unlocked**:
  - Call succeeds and the thread enters critical region;
  - **Closing** the mutex;
- If mutex is **locked**:
  - Call blocks until thread in critical region calls **mutex_unlock**
  - If multiple threads are blocked:
    - One is chosen and allowed to acquire the lock

This leads to the following solution:

```
mutex_lock:
        TSL REGISTER,MUTEX      | copy mutex to register and set mutex to 1
        CMP REGISTER,#0         | was mutex zero?
        JZE ok                  | if it was zero, mutex was unlocked, so return
        CALL thread_yield       | mutex is busy; schedule another thread
        JMP mutex_lock          | try again
ok:     RET                     | return to caller; critical region entered

mutex_unlock:
        MOVE MUTEX,#0           | store a 0 in mutex
        RET                     | return to caller
```

Figure: Implementation of mutex_lock and mutex_unlock (Source: (Tanenbaum and Bos, 2015))

Can you notice an important difference from previous examples? Any ideas

Can you notice an important difference from previous examples? Any ideas

- Previous examples relied on **busy waiting**;
- This one invokes **thread_yield** when failing to acquire the lock:
  - CPU is passed to another thread;
  - **Consequence:** No busy waiting! =)

Mutexes in Pthreads

Pthreads provides mutex functions:

Thread call	Description
Pthread_mutex_init	Create a mutex
Pthread_mutex_destroy	Destroy an existing mutex
Pthread_mutex_lock	Acquire a lock or block
Pthread_mutex_trylock	Acquire a lock or fail
Pthread_mutex_unlock	Release a lock

Figure: Some of the Pthreads calls relating to mutexes. (Source: (Tanenbaum and Bos, 2015))

Monitors

Higher-level synchronization primitive:

- Collection of procedures, variables, and data structures:
  - Grouped together in a module or package;
  - Not part of the C language!
- Processes may call the procedures in a monitor:
  - Processes cannot directly access monitor's internal data structures;

Monitors have an important property:

- **Only one** process can be active in a monitor at any instant;
- Monitor procedure calls are handled differently from other procedure calls;
- When a process calls a monitor procedure:
  - Procedure will check if any other process is active within the monitor:
    - **If so:** calling process will be suspended until other process has left the monitor
    - **If no:** other process is using the monitor, the calling process may enter.

```
monitor example
        integer i;
        condition c;

        procedure producer( );
        .
        .
        .
        end;

        procedure consumer( );
        .           .           .
        end;
end monitor;
```

Figure: A monitor example. (Source: (Tanenbaum and Bos, 2015))

How can monitors guarantee that only one process / thread will be active at any one point in time in the monitor? Any ideas?

How can monitors guarantee that only one process / thread will be active at any one point in time in the monitor? Any ideas?

- A Mutex can be used =)
- A binary semaphore can be used =)

How do monitors enable better synchronization than other alternatives?
Any ideas?

How do monitors enable better synchronization than other alternatives? Any ideas?

The monitor package is arranging for mutual exclusion:

- Instead of the programmer:
  - much less likely that something will go wrong.

Monitors also allow for **condition variables**:

- Used to determine whether or not a process / thread can continue;
- This is done through two operations: **wait** and **signal**;

Recall the producer - consumer problem (1/2):

- Producer finds the buffer full::
  - It does a **wait** on some condition variable, say, full;
  - This action causes the calling process to block;
  - Allows another process to enter the monitor, if one exists;

Recall the producer - consumer problem (2/2):

- Consumer can wake up its sleeping partner :
  - By doing a **signal** on variable full;
- **Important:** Condition variables are not counters:
  - Do not accumulate signals for later use the way semaphores do
  - if a condition variable is signaled with no one waiting on it:
    - signal is lost forever.

We can now define the **pseudocode** for the monitor:

```
monitor ProducerConsumer
        condition full, empty;
        integer count;

        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;

        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;

        count := 0;
end monitor;
```

Figure: An outline of the producer-consumer problem with monitors. Only one monitor procedure at a time is active. The buffer has N slots. (Source: (Tanenbaum and Bos, 2015))

Monitor can now be used to implement the consumer - producer problem:

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
```

```
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

What is the difference between wait/signal and sleep/wakeup? Any ideas?

What is the difference between wait/signal and sleep/wakeup? Any ideas?

Automatic mutual exclusion on monitor procedures guarantees that:

- **Producer** inside a monitor will be able to complete **wait** operation:
  - Without worrying about scheduler switching to other process / thread;
- **Consumer** will not even be let into the monitor at all until the wait is finished:
  - and the producer has been marked as no longer runnable.

What is the main conclusion you can draw from using monitors? Any ideas?

What is the main conclusion you can draw from using monitors? Any ideas?

By making the mutual exclusion of critical regions automatic:

- monitors make parallel programming much less error prone;

This concludes our study of synchronization primitives =)

Message Passing

Method of interprocess communication that uses two system calls:

- **send(destination, &message)**
  - Sends the contents of message to a given destination
- **receive(source, &message)**
  - Receives a message from a given source:
    - If no message is available: receiver can block until one arrives;
    - Alternatively, receiver can return immediately with an error code;
- Destination and source fields can be specified using other OS system calls;

Scheduling

Lets have a more detailed look at scheduling:

First, what is the oficial definition of scheduling? Any ideas?

Scheduling

Lets have a more detailed look at scheduling:

First, what is the oficial definition of scheduling? Any ideas?

- Normal for computers to execute multiple processes / threads
- Who to choose to execute next when multiple processes / threads exist in ``ready'' state?
- **Scheduler:** OS part that makes this choice:
  - According to some **scheduling algorithm**;

Remember this?

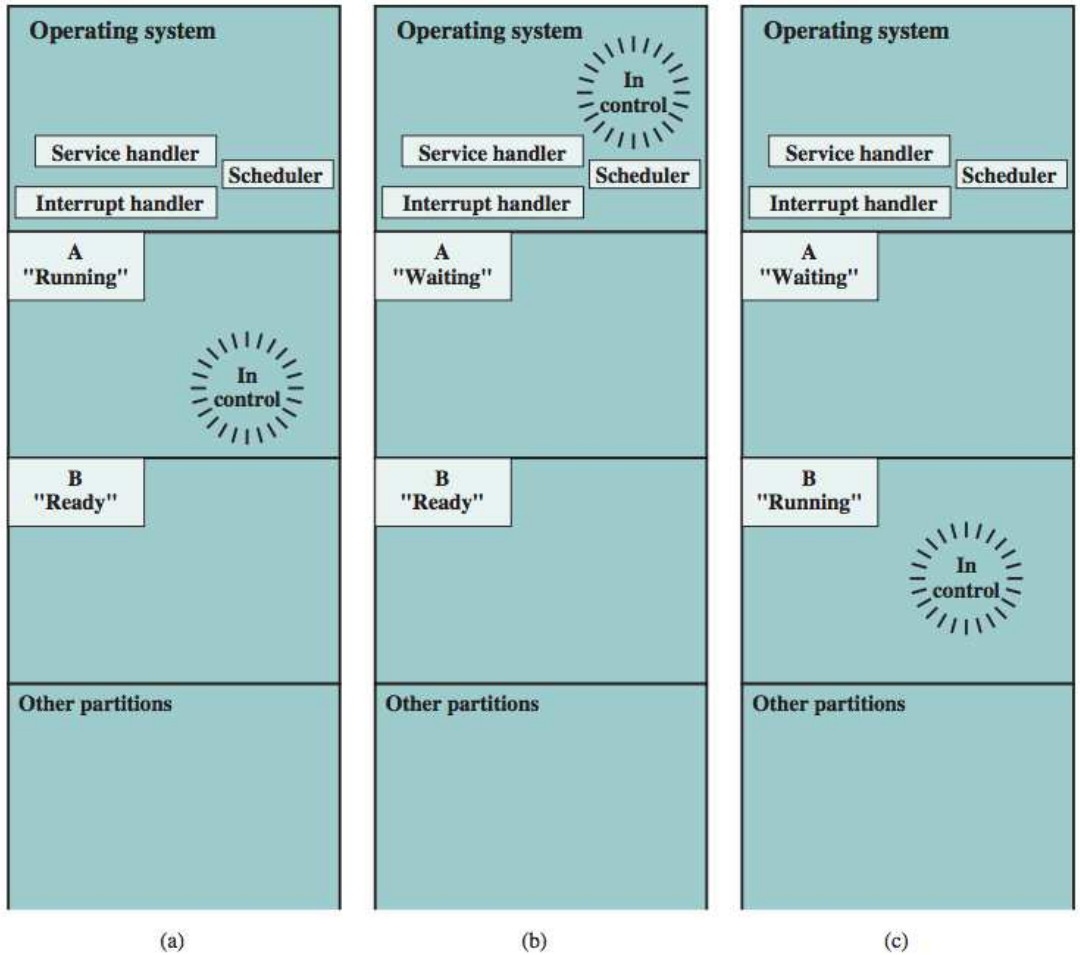

Figure: Scheduling Example (Source: (Stallings, 2015))

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

Possibility 1: When a new process is created.

- Who should run? The parent? Or the child?
  - Both processes are in **ready** state;
- Scheduler can legitimately choose to run either process;

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

Possibility 2: When a process exits.

- Who should run next?
  - Some other process must be chosen from the set of **ready** processes;

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

Possibility 3: When a process blocks on I/O, semaphore or for other reason.

- Who should run next?
  - Some other process must be chosen from the set of **ready** processes;

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

Possibility 4: When an I/O interrupt occurs.

- Who should run next?
  - Some other process that was **blocked** waiting for the I/O may now be **ready**.
  - The process that was running at the time of the interrupt?
  - Or some other third process?
- Scheduler can legitimately choose any of these processes;

When to schedule

When should the scheduler make scheduling decisions? Any ideas?

Possibility 5: When a process / thread exceeds its **time**.

- Who should run next?
  - Some other process from the **ready** set.
- Scheduler can legitimately choose any of these processes;

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

Depends on the system requirements:

- **All computational systems**
- **Interactive systems**
- **Real-time systems**

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

All computational systems: (1/3)

- Fairness - giving each process a fair share of the CPU:
  - However, different categories of processes may be treated differently.
  - *E.g.:* Minecraft vs. nuclear reactor's computer;

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

All computational systems: (2/3)

- Policy enforcement - seeing that stated policy is carried out.
  - More vs. Less priority processes / threads

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

All computational systems: (3/3)

- CPU utilization - keep the CPU busy all the time.
  - Increase CPU efficiency.

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

Interactive Systems:

- Response time - respond to requests quickly
  - Minimize response time to user's request;
  - Processing interactive requests first will be perceived as good service;

Scheduling Algorithm Goals

What do you think are the main goals of the scheduling algorithm?

Real time systems:

- Meeting deadlines - avoid losing data
  - **E.g.:** the processes / threads handling aircraft's sensors;

Scheduling

A lot of scheduling techniques were developed over time:

- round-robin scheduling;
- priority scheduling;
- multiple queues;
- etc;

As we have seen previously:

- Linux Kernel 2.6 implements the CFS scheduler;

Linux Kernel 2.6 implements the CFS scheduler (1/3):

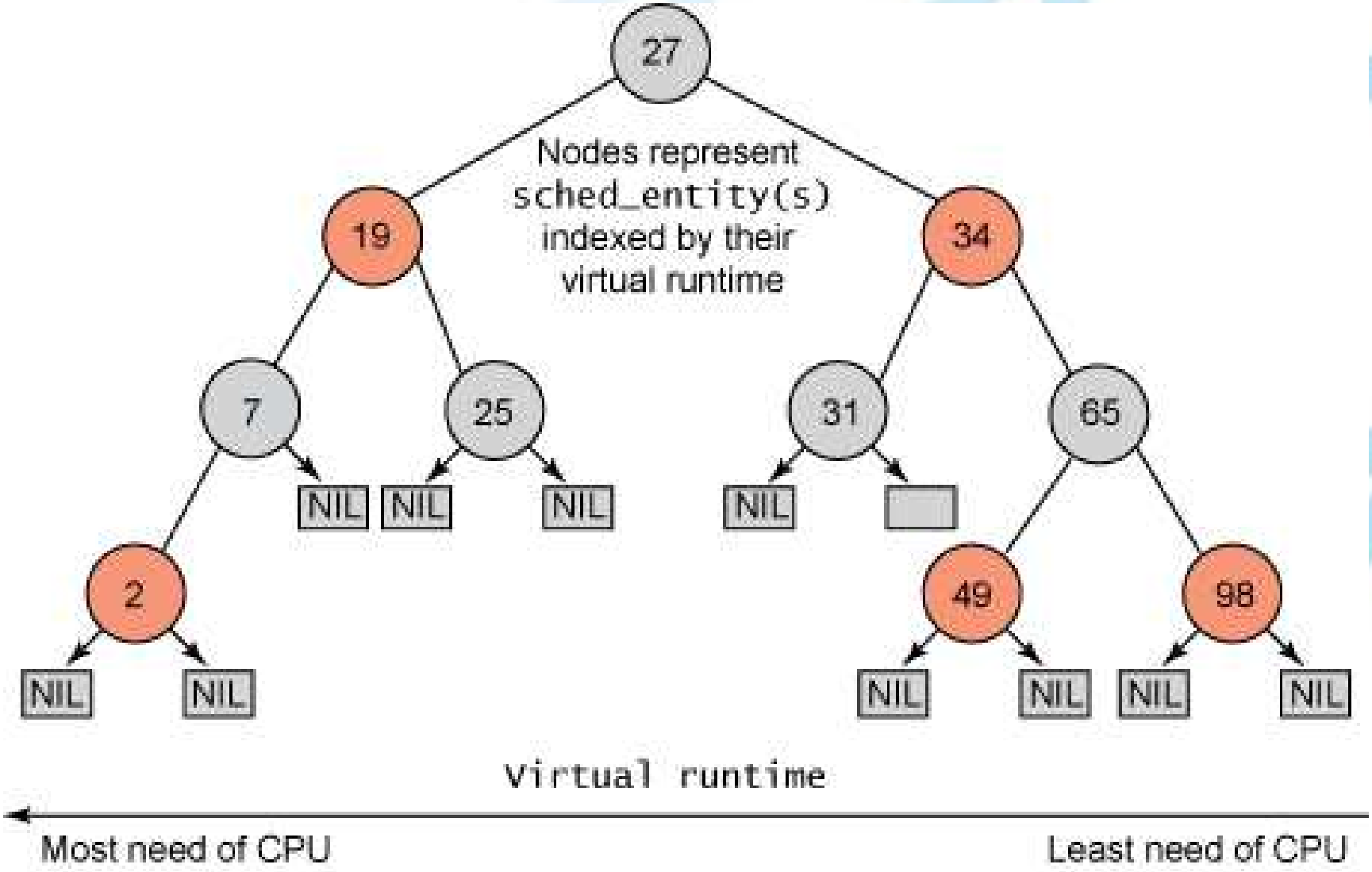

Figure: (Source: IBM)

Linux Kernel 2.6 implements the CFS scheduler (2/3):

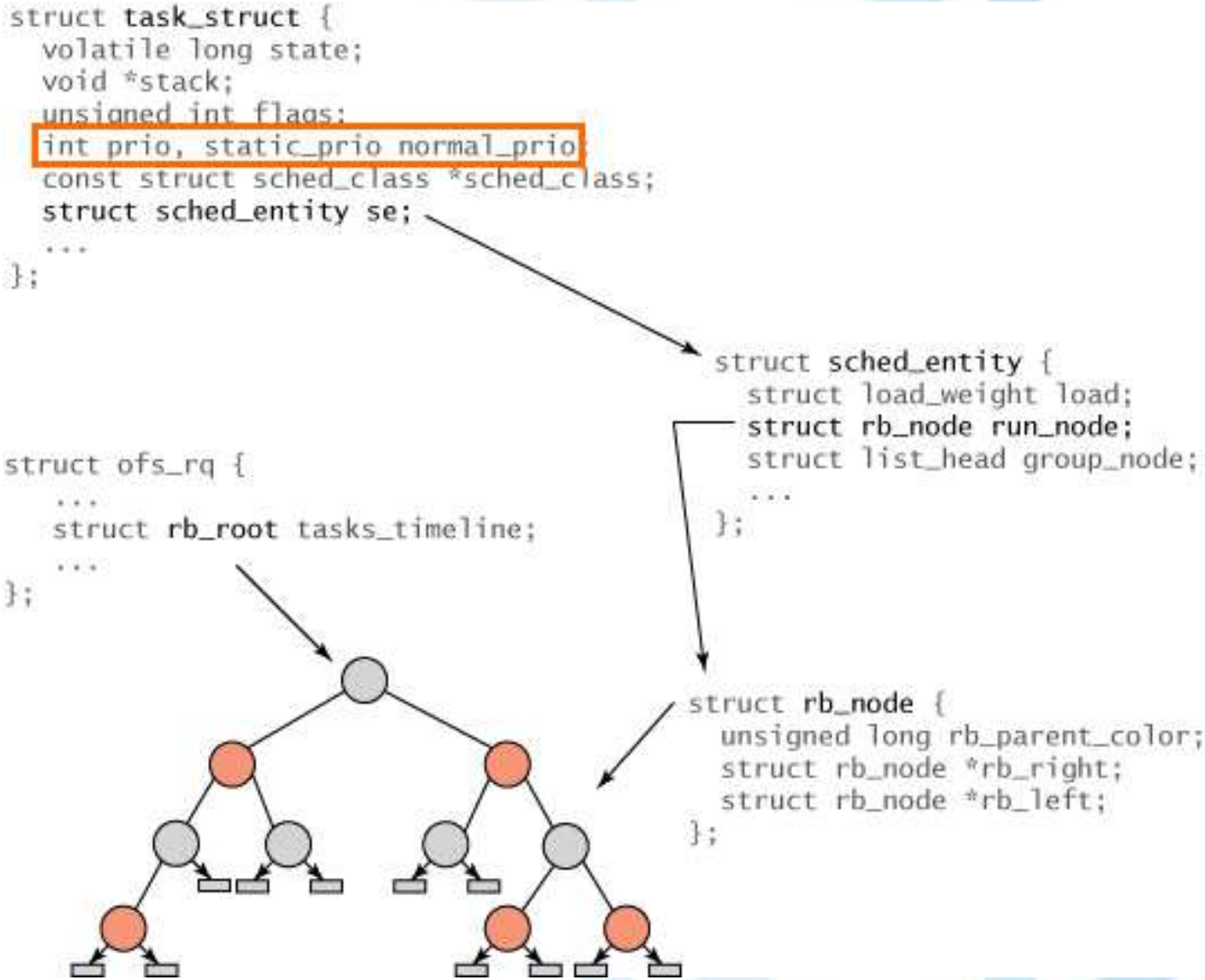

Figure: (Source: IBM)

Linux Kernel 2.6 implements the CFS scheduler (1/3):

- The virtual_runtime is used to determine who should be executed next:
  - Higher priority processes will be allowed to execute:
    - for a longer time;
    - more frequently;
  - Lower priority processes will be allowed to execute:
    - for a smaller time;
    - less frequently;

Classical IPC Problems

OS literature is full of interesting synchronization problems, *e.g.:*

- The Dining Philosophers Problem;
- The Readers and Writers Problem;

Guess what we will be seeing next ;)

The Dining Philosophers Problem

Five philosophers are seated around a circular table:

- Each philosopher has a plate of spaghetti;
- The spaghetti is so slippery that a philosopher needs two forks to eat it;
- Between each pair of plates is one fork.

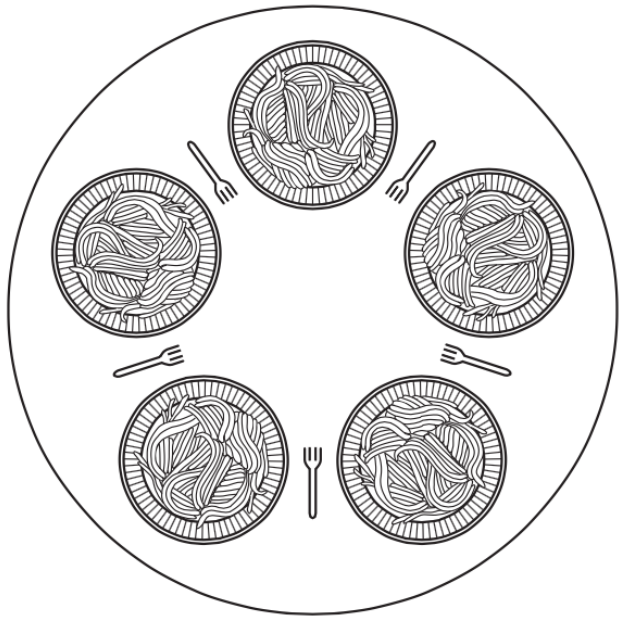

Figure: (Source: (Tanenbaum and Bos, 2015))

Philosopher alternates between thinking and eating.

- When a philosopher gets sufficiently hungry:
  - Tries to acquire her left and right forks, one at a time
  - If successful in acquiring two forks:
    - eats for a while then puts down the forks;

Can you write a program for each philosopher that does what it is supposed to do and never gets stuck?

The **obvious** solution:

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Figure: (Source: (Tanenbaum and Bos, 2015))

Can you see anything wrong with the previous solution? Any ideas?

Can you see anything wrong with the previous solution? Any ideas?

Suppose that all five philosophers take their left forks simultaneously:

- None will be able to take their right forks
- This will result in a **deadlock**

Can you see anything wrong with the previous solution? Any ideas?

Suppose that all five philosophers take their left forks simultaneously:

- None will be able to take their right forks
- This will result in a **deadlock**

How can we circumvent this problem? Any ideas?

Obtaining maximum parallelism for N philosophers:

- Each philosopher's **state** needs to be maintained:
  - eating, thinking, or hungry (trying to acquire forks);
- Philosopher may move into eating state only if **neither** neighbour is eating;
- An array of semaphores, one per philosopher, is needed:
  - so hungry philosophers can block if the needed forks are busy;

The **revised** solution:

```
semaphore s[N];                          /* one semaphore per philosopher */

void philosopher(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        while (TRUE) {                   /* repeat forever */
                think( );                /* philosopher is thinking */
                take_forks(i);           /* acquire two forks or block */
                eat( );                  /* yum-yum, spaghetti */
                put_forks(i);            /* put both forks back on table */
        }
}

void take_forks(int i)                   /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                    /* enter critical region */
        state[i] = HUNGRY;               /* record fact that philosopher i is hungry */
        test(i);                         /* try to acquire 2 forks */
        up(&mutex);                      /* exit critical region */
        down(&s[i]);                     /* block if forks were not acquired */
}

void put_forks(i)                        /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                    /* enter critical region */
        state[i] = THINKING;             /* philosopher has finished eating */
        test(LEFT);                      /* see if left neighbor can now eat */
        test(RIGHT);                     /* see if right neighbor can now eat */
        up(&mutex);                      /* exit critical region */
}

void test(i) /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Readers and Writers Problem

Example:

- Airline reservation system:
  - with many competing processes wishing to read and write it.
- Multiple processes may be reading the database at the same time:
  - **However:** if one process is updating the database:
    - no other processes may have access to the database, not even readers

The question is how do you program the readers and the writers?

The **obvious** solution:

```
typedef int semaphore;                  /* use your imagination */
semaphore mutex = 1;                    /* controls access to rc */
semaphore db = 1;                       /* controls access to the database */
int rc = 0;                             /* # of processes reading or wanting to */

void reader(void)
{
    while (TRUE) {                      /* repeat forever */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc + 1;                    /* one reader more now */
        if (rc == 1) down(&db);         /* if this is the first reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        read_data_base( );              /* access the data */
        down(&mutex);                   /* get exclusive access to rc */
        rc = rc – 1;                    /* one reader fewer now */
        if (rc == 0) up(&db);           /* if this is the last reader ... */
        up(&mutex);                     /* release exclusive access to rc */
        use_data_read( );               /* noncritical region */
    }
}


void writer(void)
{
    while (TRUE) {                      /* repeat forever */
        think_up_data( );               /* noncritical region */
        down(&db);                      /* get exclusive access */
        write_data_base( );             /* update the data */
        up(&db);                        /* release exclusive access */
    }
}
```

In this solution:

- First reader to get access to the database does a **down(&db)**;
- Subsequent readers merely increment a counter **rc**;
- As readers leave:
  - They decrement the counter;
  - Last reader to leave does an up on the semaphore:
    - Allowing a blocked writer, if there is one, to get in.

Can you see any problem with the previous solution? Any ideas?

Can you see any problem with the previous solution? Any ideas?

While there is a reader other readers my be **admitted**:

- As long as at least one reader is still active:
  - A **writer** will never be allowed to work

What can we do to solve this problem? Any ideas?

What can we do to solve this problem? Any ideas?

A reader arrives and a writer is waiting:

- Reader is suspended behind the writer:
  - Instead of being admitted immediately;
- In this way:
  - Writer has to wait for readers that were active when it arrived to finish...
  - ...but does not have to wait for readers that came along after it

References I

Stallings, W. (2015).

Computer Organization and Architecture.

Pearson Education.

Tanenbaum, A. and Bos, H. (2015).

Modern Operating Systems.

Pearson Education Limited.