BART, R and Operating Systems (OS)

- In association with our collaborators, we have created several R packages for BART
- GNU R was started by Ross Ihaka and Robert Gentelman as a successor to Bell Labs S that was only available on UNIX
- 1993: R first released for Apple MacOS (classic), but ports to Microsoft Windows, UNIX and GNU Linux soon followed
- R does its best to treat the three modern platforms equally: Windows, UNIX/Linux and Apple macOS (OS X)
- But, there are really just two OS types as far as R is concerned
- `R> .Platform$OS.type`
  `"unix"` for UNIX/Linux/macOS and `"windows"` for Windows
- However, there are some fundamental differences that R cannot address: in particular, multi-threading
- We support BART on all R platforms but Windows is the most challenging: we have workarounds for some issues

BART software supporting S3 `predict` with URLs

		R packages		
Debut	Language	Stable (CRAN)	Development	Multi-threading
2006	C++	**BayesTree**	None	None
2013	Java	**bartMachine**		Java
2014	C++	**dbarts**		forking
2014	C++	MPI BART source code		MPI
2017	C++	**BART** 2.9.4*	**BART3***	OpenMP/forking
2019	C++	**rbart** 1.0*	**hbart***	OpenMP
2019	C++	None	**mxBART***	OpenMP/forking
2021	C++	None	**mBART***	OpenMP/forking
2021	C++	**nftbart** 1.6*	**nftbart***	OpenMP
		*Descendents of MPI BART		

Development on `github.com` by users `rsparapa` (me), `cspanbauer` (Charley Spanbauer) and `remcc` (Rob McCulloch)
Special thanks to Rob (**BART**), Matt Pratola for **rbart**
Hugh Chipman, Robert Gramacy, the R Core team,
the Rcpp Core team and so many others in the FOSS community!

BART software features: descendents of MPI BART

Stable Development	**BART** **BART3**	**nftbart** **nftbart**	**rbart** **hbart**	 **mBART**
`github.com` user		`rsparapa`		`remcc`
`predict` function	Yes	Yes	Yes	**BART**
heteroskedastic	No	Yes	Yes	No
monotonic	No	No	No	Yes
continuous	Yes	Yes	Yes	Yes
binary/categorical	Yes	No	No	No
right censoring	Yes	Yes	No	No
left censoring	No	Yes	No	No
competing risks	Yes	No	No	No
recurrent events	Yes	No	No	No
sparse prior	Yes	No	No	No
marginal effects	**BART3**	Yes	No	No
missing imputation	Yes	Yes	No	No
advanced tree proposals	No	Yes	Yes	No
nonparametric error	No	Yes	No	No
C++ header-only	**BART3**	No	**hbart**	No

Skeleton of the **BART/BART3** R package

Directory	File Example	Description
root	`configure`	To dectect OpenMP for `"unix"`
	`DESCRIPTION`	Dependency on **Rcpp** and others
`R`	`gbart.R`	*Generalized* BART function
	`wbart.R`	*Weighted* BART function
	`predict.pbart.R`	predict for `"pbart"`/probit type
	`predict.wbart.R`	predict for `"wbart"`/continuous type
`data`	`lung.rda`	Advanced lung cancer data
`demo`	`boston.R`	Boston housing demo
	`lung.surv.bart.R`	Advanced lung cancer demo
`man`	`gbart.Rd`	Help pages
	`wbart.Rd`	
	`predict.pbart.Rd`	
	`predict.wbart.Rd`	
`src`	`Makevars`	Hard-wired settings for `"windows"`
	`Makevars.in`	`configure` OpenMP template for `"unix"` `Makevars` file

BART and multi-threading

- Multi-threading is supported by software frameworks such as OpenMP and the Message Passing Interface (MPI)
- MPI can be employed for both simple multi-threading and for distributed computing, e.g., MPI software initially written for a single system could be extended to operate on multiple systems as computational needs expand
- For MPI, BART software was re-written with C++ objects simple to modify/maintain for distributed computing: we call this the MPI BART code (Pratola et al. 2014, JCGS)
- The **BART/BART3** and **rbart/hbart/nftbart** packages are all descendants of MPI BART and its programmer-friendly objects, but we have moved on from MPI mainly to OpenMP
- For a brief primer on R, BART and multi-threading go to slide 27

Testing multi-threading after installing **BART/BART3**

- `parallel::detectCores`
- Returns the number of threads that the computer is capable of
- The number of *threads* rather than the number of *cores* since they are not necessarily one-to-one
- For example, on my desktop, I have 1 CPU with 6 cores and `detectCores` returns 12
- `BART::mc.cores.openmp/BART3::mc.cores.openmp`
- Returns whether OpenMP has been detected
  >0 (Yes) vs. 0 (No)

BART and multi-threading

- Multi-threading is supported in two ways
  1) via the parallel package and 2) via OpenMP
- OpenMP takes advantage of modern hardware by performing multi-threading on single machines which often have multiple CPUs each with multiple cores
- **BART/BART3** only use OpenMP for parallelizing `predict` function calculations
- **rbart/hbart/nftbart** use OpenMP for fitting and predicting
- OpenMP support is *detected* at package installation by the `configure` script on UNIX/Linux/macOS that defines a C pre-processor macro called `_OPENMP` if available
- But a `configure` script can't run on Windows
- **BART/BART3/nftbart** *hard-wired* for Windows OpenMP
- In `src/Makevars`, Windows compiler switches for OpenMP (add to any *source* package needing OpenMP on Windows)

```
PKG_CXXFLAGS = -fopenmp
PKG_LIBS = -fopenmp
```

Installation resources for R and R packages: BEWARE

- The Comprehensive R Archive Network (CRAN) `http://cran.r-project.org` has R binaries for Windows, macOS and many flavors of Linux
- CRAN is a wealth of manuals, advice, FAQs, etc.
- Avoid the pitfalls: just do it the "CRAN way"!
- Do NOT use *package managers* unless CRAN approves
- Extra Packages for Enterprise Linux (EPEL) is approved for Red Hat-flavored Linux
- And, so are Debian-flavored packages at `debian.org`
- But, EPEL and Debian are exceptions
- For example, on macOS, the Homebrew and conda package managers are NOT approved
- Only use CRAN binaries and/or build with CRAN approved tool chains!
- Be safe, not sorry

Installation resources for R and R packages

- Windows Rtools 4.3 `https://cran.r-project.org/bin/windows/Rtools/rtools43/rtools.html`
  mainly, the GNU Compiler Collection (GCC) v. 12
- macOS tools: `https://mac.r-project.org/tools`
  for BART, we need Xcode installed from the App Store
  and the command-line tools which are installed as follows
  `terminal$ sudo xcode-select --install`
  with OpenMP at `https://mac.r-project.org/openmp`
- **remotes** package
  `https://cran.r-project.org/package=remotes`
- **Rcpp** package
  `https://cran.r-project.org/package=Rcpp`

Installing R packages from source

- Installing R packages from source needs a compiler tool chain that support **Rcpp** and various BART packages
  therefore, we need ISO standard C++11 (2011) or higher
- CRAN now defaults to ISO standard C++17 (2017)
  with C++11 or ISO standard C++14 (2014) optional
- So a CRAN compatible C++ compiler is needed
  there are two common *flavors* used by CRAN
  the GNU Compiler Collection (GCC) and LLVM Clang
  Clang maintains compatibility with GCC
  (but a Fortran compiler is NOT needed for BART)
- For Windows, CRAN R Tools provide GCC with OpenMP
  `https://cran.r-project.org/bin/windows/Rtools/rtools43/rtools.html`
- For macOS, rely on Apple Xcode's Clang
  but you have to install Clang's OpenMP library from CRAN
  for more details see next slide

Auto-installing OpenMP on macOS with `configure`

- Get the tarball from `https://mac.r-project.org/openmp`
- The latest version of the OpenMP library (as of this writing) is 14.0.6 for Xcode 14.3 (Apple clang 14.0.3)
- Manually install it from the `~/Downloads` folder compressed

```
$ sudo bash
$ tar fvxz openmp-14.0.6-darwin20-Release.tar.gz -C /
```

or uncompressed

```
$ tar fvx openmp-14.0.6-darwin20-Release.tar -C /
```

- For example, install **nftbart**
  `$ R CMD INSTALL nftbart_1.6.tar.gz`
- Then you should see the following if OpenMP is auto-detected

```
checking for clang++ ...  option to support OpenMP...
-Xlinker -lomp -Xclang -fopenmp
```

- Due to `-lomp` which is needed for linking only, you will see a harmless warning when compiling (linking is fine too)
  `clang: warning: -lomp: 'linker' input unused [-Wunused-command-line-argument]`

Installing R packages

- The variable `.Library` contains the location of the default directory for R packages
- `R> .Library`
- Depending on the OS, this directory may not be writeable
- To create an alternative library for your R packages that you can edit, use the `.libPaths()` function
- `R> .libPaths('~/RLIB')`
- But you need to create the directories obviously before installing
- `terminal$ mkdir ~/RLIB`
- Similarly, you can find where any R package is installed with `system.file()`
- `R> system.file(package='BART')`
- For example, to find the `demo` directory
- `R> system.file('demo', package='BART')`

Installing R packages with CRAN

- CRAN has 19124 R add-on packages as of this writing (01/28/23)
  there will be many more by the time you read this
- To install an R package from CRAN
  The two most reliable, and likely complete, mirrors I use
  `http://lib.stat.cmu.edu/R/CRAN` at Carnegie-Mellon and
  `http://cran.wustl.edu` at Washington University in St.L.
  N.B. http NOT https

```
R> options(repos=c(CRAN="http://lib.stat.cmu.edu/R/CRAN"))
R> install.packages("remotes", dependencies=TRUE)
R> install.packages("Rcpp", dependencies=TRUE)
R> install.packages("BART", dependencies=TRUE)
R> install.packages("nftbart", dependencies=TRUE)
```

To install all CRAN packages (takes hours: we run this over-night)

```
R> install.packages(available.packages()[ , 1])
```

Some of them will fail for missing system dependencies like device drivers, required software, etc., but R will try to install them all

Installing R packages with Bioconductor

- The Bioconductor Project produces R packages for bioinformatics: `http://bioconductor.org`
- Bioconductor versions are tied to specific R versions
  `R> tools:::.BioC_version_associated_with_R_version()`
  for example, the return value is `"3.12"` with R 4.0.4
- To install the package named `limma` (and R or Bioconductor package dependencies, if any)
  `R> source("http://bioconductor.org/biocLite.R")`
  `R> biocLite("limma")`
- To install all Bioconductor packages (takes a while):
  `R> biocLite(all_group())`

build and INSTALL R packages: command line

- For macOS/Linux, use `bash`
- For Windows, use `CMD.EXE`
- Build and install R packages from the command line: `$`
- This works with your own R packages or those of others
- If it is your own in the sub-directory `PACKAGE`, then build it:
  `$ R CMD build PACKAGE`
- For others, download the archive of source files
  either a gzipped `TARFILE` ending in `.tar.gz` or `.tgz`
  or a PKWARE/Info-ZIP `ZIPFILE` ending in `.zip`
- Unpack it: `$ tar xzf TARFILE` or `$ unzip ZIPFILE` which should create the `PACKAGE` sub-directory
- Build the package: `$ R CMD build PACKAGE`
- Typically the vignettes take a long time or may crash the build
  `$ R CMD build --no-build-vignettes PACKAGE`
- So now you have created `PACKAGE_VERSION.tar.gz`
- Install it: `$ R CMD INSTALL PACKAGE_VERSION.tar.gz`
- And you can remove it later: `$ R CMD REMOVE PACKAGE`

`build` and `INSTALL` R packages: `remotes` package

- You can build and install R packages from anywhere on the internet with the `remotes` package
- For example, former CRAN packages that have been Archived: `https://cran.r-project.org/src/contrib/Archive`
- These can be installed with the `install_url` function
- Or R packages on `https://github.com`
- These can be installed with the `install_github` function
- However, R 3.6.2 or higher appears to be necessary
- For example, the **BART3** package (beta **BART**) at `https://github.com/rsparapa/bnptools/tree/master/BART3`
- `R> install_github("rsparapa/bnptools/BART3")`
- Or the mBART package, monotonic BART, at `https://github.com/remcc/mBART_shlib/tree/main/mBART`
- `R> install_github("remcc/mBART_shlib/mBART")`
- N.B. installing from the command line is much faster

build and INSTALL R packages with git

- This is much faster than `remotes::install_github`
- To install either R package: **BART3** or **mBART**
  first, you have to "clone" the repository

```
$ mkdir DIR
$ cd DIR
$ git clone https://github.com/rsparapa/bnptools.git
$ cd bnptools ## where BART3 is a sub-directory
$ R CMD build --no-build-vignettes BART3
$ R CMD INSTALL BART3_VERSION.tar.gz
$ cd ..
$ git clone https://github.com/remcc/mBART_shlib.git
$ cd mBART_shlib ## where mBART is a sub-directory
$ R CMD build --no-build-vignettes mBART
$ R CMD INSTALL mBART_VERSION.tar.gz
```

Intelligent development environments (IDE) for R/C++

- To work with BART, you need an IDE for R
- And, if you need to tinker with BART, you also need C++
- RStudio is a popular IDE, but it ONLY does R
- And, it requires that R be built with `--enable-R-shlib`
- But, that will prevent the GNU debugger, `gdb`, from working
- The debugger is great technology that we refuse to give up!

Emacs and ESS for R/C++

- 1975: Emacs "Editor MACroS" by Richard Stallman (RMS)
  intelligent development environment (IDE) for programmers
- 1980: US law changes to recognize software Copyright
- 1983: UniPress starts selling "Gosling version" of Emacs
  RMS founds the GNU project
  GNU stands for "GNU is Not UNIX"
  "a complete UNIX-compatible software system"
- 1984: RMS releases GNU Emacs as free software
  re-written in C with Elisp (Emacs Lisp) for modes
- 1986: emacs FORTRAN-mode: IDE for FORTRAN
- 1989: the GNU General Public License (GPL) for free software
- 1994: Anthony Rossini releases ESS (GPL) containing
  Emacs modes for statistical software like ESS[R]

Installing Emacs/ESS for your R IDE

- Vincent Goulet's *Modified* Emacs installable binaries for both Windows and macOS with ESS and other goodies
  many modes for programming like C/C++ and markup such as AUCTeX: a LaTeX support mode
  English, French, German and Spanish dictionaries for Hunspell
  `http://hunspell.github.io`
- For Windows:
  `https://vigou3.gitlab.io/emacs-modified-windows`
- For macOS:
  `https://vigou3.gitlab.io/emacs-modified-macos`
- Check ESS is working with `M-x ess-version`
- For macOS, the Modified Emacs app is crash-prone
- Homebrew is a macOS and Linux package manager
  but its compiler tool chain is NOT compatible with R
- However, you can install the very stable Homebrew Emacs binaries without the compiler baggage
- So install Homebrew Emacs and clone the Modified setup

Installing Emacs/ESS for macOS

0. macOS 13 (Ventura): in "System Settings" under "Privacy & Security" give Terminal permission for "App Management"
1. Install latest macOS Modified binary from
   `https://vigou3.gitlab.io/emacs-modified-macos`
   Launch it and run `M-x ess-version`
   Rename it to `/Applications/EmacsMod.app`
   Currently, this is Emacs 28.1 (as of this writing)
2. Install the Homebrew emacs binary from
   `https://github.com/railwaycat/homebrew-emacsmacport/releases`
   Download emacs-EMACSv-mac-RELv-OSv.zip
   e.g., EMACSv=28.2, RELv=9.1, OSv=12.6
   Emacs 28.2 is the latest version (as of this writing)
   OS 12.6 is a Monterrey update from September 2022
   Copy `Emacs.app` with the Terminal
   `$ cp -r ~/Downloads/Emacs.app /Applications`
   Launch Emacs and then exit before proceeding to next step

Installing Emacs/ESS for macOS

3. Create the `site-lisp` directory from the Terminal

```
$ sudo bash                    ## start a shell as superuser
$ MOD=/Applications/EmacsMod.app/Contents/Resources/lisp
$ HB=/opt/homebrew/share/emacs/site-lisp
$ mkdir -p $HB                 ## your new site-lisp library
$ cp -r ${MOD}/* $HB           ## copy the goodies
$ chown -R root:wheel $HB
$ chmod -R 775 $HB
$ exit                         ## exit from superuser shell
```

4. Set Apple Human Interface Guidelines Apple-key definitions
   copy Command-c, cut Command-x, paste Command-v, etc.
   Copy `emacs-macos.el` to your user emacs settings `~/.emacs`

```
$ mkdir DIR
$ cd DIR
$ git clone https://github.com/rsparapa/bnptools.git
$ cp bnptools/emacs-macos.el ~/.emacs
```

Installing ESS with `git`

Regardless of your platform, you may need to install ESS from source to get the latest version/bug-fixes/etc.

1. Clone ESS with `git`
   ```
   $ git clone https://github.com/emacs-ess/ESS.git
   $ cd ESS
   ```
2. Edit the file `Makeconf` to match your emacs setup
3. Build ESS
   ```
   $ nohup make all >& all.txt &
   ```
4. And install it
   ```
   $ make install
   ```

Welcome to Emacs

- Modifier Keys: Emacs documentation looks like this
- `C-KEY` means hold down the Control key while pressing `KEY`
- For example, `C-x` means hold down Control while pressing `x`
- `M-KEY` means hold down the Meta key while pressing `KEY`
- On PC, the Meta key is usually the Alt key
- On Mac, the Meta key is Option (from `emacs-macos.el`)
  In XQuartz Preferences: "Option keys send Alt_L and Alt_R"
- Or, you can press `Esc`, release, and then press `KEY`
- Execute an emacs command: `M-x COMMAND` which is followed by pressing `Enter`
- Check ESS is working with `M-x ess-version`
- For example, `M-x man` to bring up a man page
  or `M-x info` the directory of info pages
- `S-KEY` means hold down the Shift key while pressing `KEY`

Common Emacs Shortcuts

- `C-h` is the help key and `F1` is its alias
- But you have to get your laptop to generate an `F1`
  on PC/Mac, check your keyboard settings for function keys
- For example, `C-h k` describes the next key pressed
- Try `C-h k F1 k`
- Interrupt command: `C-g`
- Save the file: `C-x C-s`
- Quit emacs: `C-x C-c`
- `C-x C-f` is open a file or a directory
- `F2` is refresh (ESS)
- `F8` is go to `*shell*` buffer (ESS)
- `M-w` is copy
- `C-y` is paste
- `C-w` and `Delete` are cut

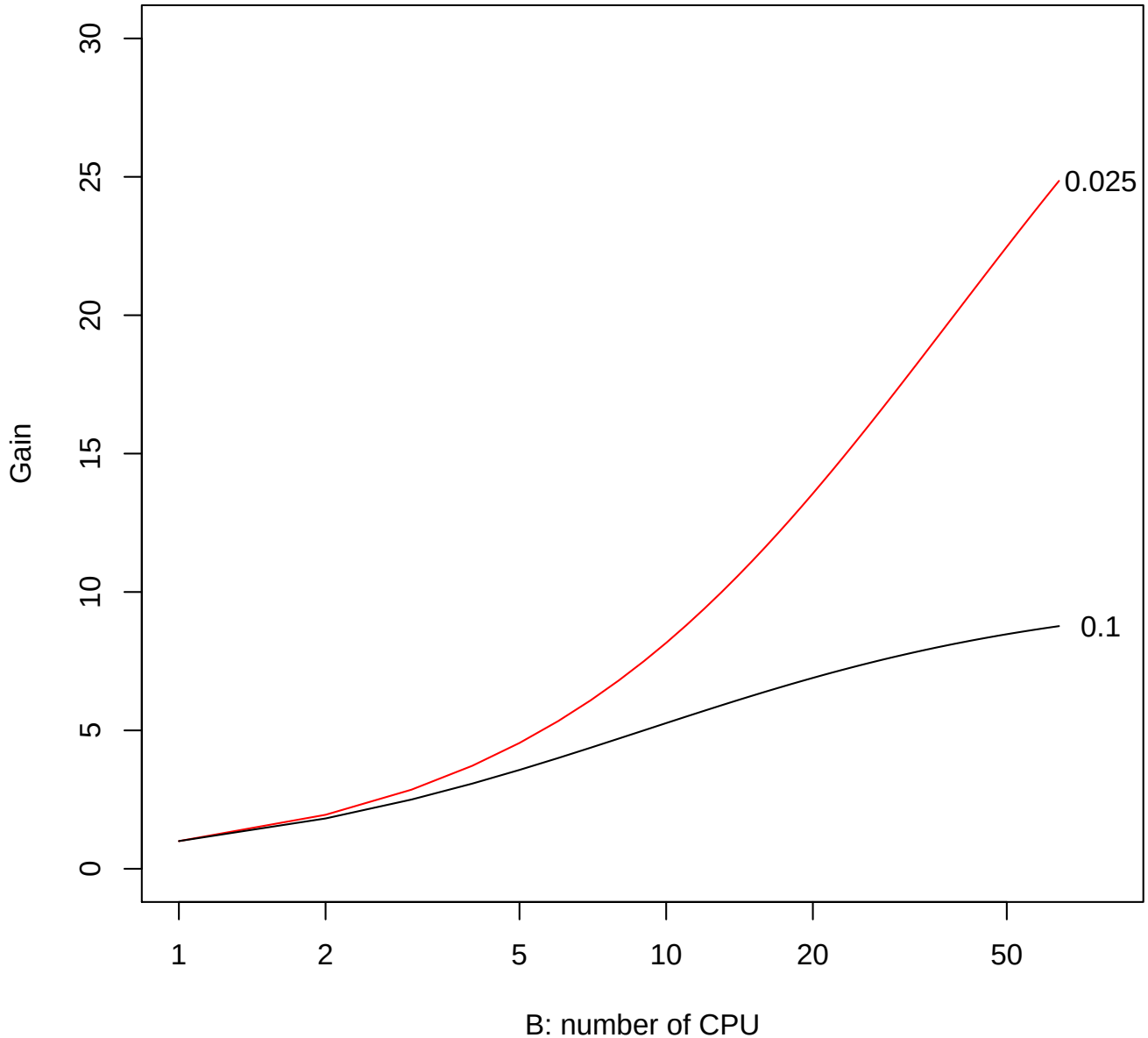
Common Emacs Shortcuts

- `C-c` comments a region (an area of text selected)
- `C-u` is the prefix command so `C-u C-c` uncomments a region
- `C-x 2` splits the buffer top over bottom
- `C-x 1` unsplits the buffer
- `C-x 3` splits the buffer left and right
- `C-s` starts a forward search
- Repeating `C-s` searches for the same string again
- `C-r` starts a reverse search
- `C-u C-s` starts a forward regular expression search
- See Search:Regexps entry of emacs manual : `M-x info`

Multi-threading and symmetric multi-processing

- Multi-threading and symmetric multi-processing are advanced technology that are surprisingly easy to use today
- Today, most off-the-shelf hardware available features 1 to 4 CPUs each of which is capable of multi-threading
- For example, on my desktop, I have 1 CPU with 6 cores capable of 12 threads (2 threads/core)
- Multi-threading emerged quite early in the digital computer era with the groundwork laid way back in the 1960s
- In 1962, Burroughs released the D825 which was the first commercial hardware capable of symmetric multiprocessing (SMP) with CPUs
- In 1967, Gene Amdahl derived the theoretical limits for multi-threading which came to be known as Amdahl's law
- If B is the number of CPUs and b is the fraction of work that can't be parallelized, then the gain due to multi-threading is $((1-b)/B+b)^{-1}$

Amdahl's law: $((1-b)/B+b)^{-1}$ where $b \in \{0.025, 0.1\}$



Multi-threading with `parallel` package

- The `mcparallel` function uses *forking* to facilitate multi-threading (forking is NOT available on Windows)
- *Fork* is an operation where a process creates a copy of itself
- A *forked* R *child* process has memory address *pointers* to all of the objects known to the *parent* such as loaded packages, function definitions, data frames, etc.
- But, these *shared* objects are NOT copied into memory for each child: that would be a huge waste of resources!
- Each child has a memory address *pointer* to these objects
- Furthermore, R has a *copy on write* philosophy
- If a child writes to an object owned by the parent, a copy is made for the child while the parent retains the original
- This is convenient, but can be dangerous with multiple threads
- For example, if this is a big object, now that object has multiple instances which might consume a lot of memory

The `mcparallel` function and nice

```
R> library(parallel) ## an example of multi-threading
R> library(tools)
R> for(i in 1:mc.cores)
R>    mcparallel({psnice(value=19); expr})
R> obj.list = mccollect()
...
```

- `expr` is processed `mc.cores` times each in their own threads

Paraphrasing the `psnice` documentation
Unix schedules processes to execute according to their priority. Priority is assigned values from 0 to 39 with 20 being the normal priority and (counter-intuitively) larger numeric values denoting lower priority. Adding to the complexity, there is a *nice* value: the amount by which the priority exceeds 20. Processes with higher nice values will receive less CPU time than those with normal priority. Generally, processes with nice value 19 are only run when the system would otherwise be idle to enhance system interactivity.

The `mccollect` function

- `mccollect` returns a list of return values from each thread
- in my experience, these are returned last in, first out (LIFO) the reverse from what we might have expected
- occasionally, a sporadic failure in one, or more, of the threads failed component(s) are missing from the list of return values
- if it is sporadic: re-running without any changes can succeed
- `class(obj)[1]!=type` is likely an error message so return it

```
R> obj.list = mccollect() ## last in, first out
R> obj = obj.list[[1]]
R> if(mc.cores==1 | class(obj)[1]!=type) {
R>    return(obj)
R> } else {
R>    m = length(obj.list)
R>    if(mc.cores!=m)
R>      warning(paste0("The number of items is only ", m))
    ...
R> }
```

The `mcparallel` function and random number generation

- We want each thread to have its own *stream* of random numbers that is reproducible
- There is a special random number generator for this purpose
- L'Ecuyer's combined multiple-recursive generator (CMRG)

```
R> library(parallel)
R> library(tools)
R> RNGkind("L'Ecuyer-CMRG")
R> set.seed(seed)
R> mc.reset.stream()
R> for(i in 1:mc.cores)
R>    mcparallel({psnice(value=19); expr})
```