NTNU | Norwegian University of Science and Technology

Operating Systems

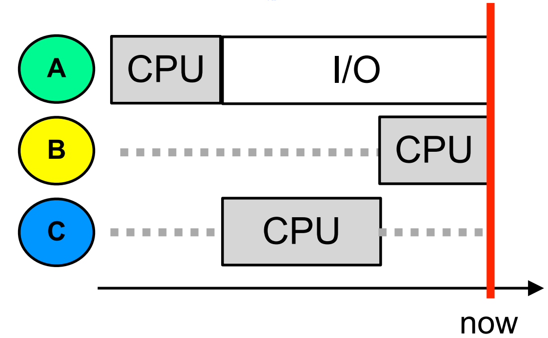
Lecture 4: Processes

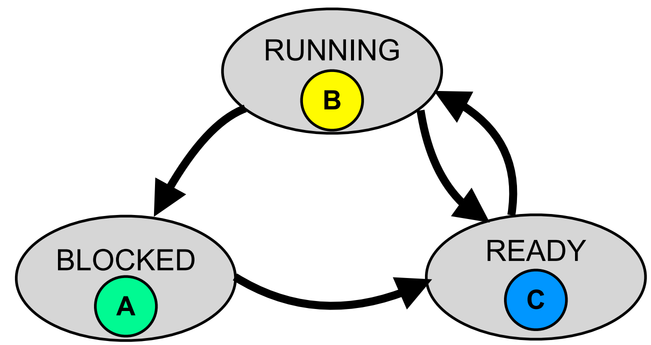
Michael Engel

Review: processes…

Processes

- are "programs in execution"
  - dynamic, not static
  - alternating sequences of "CPU bursts" and "I/O bursts"
- require resources of the computer
  - CPU, memory, I/O devices
- have a state
  - READY, RUNNING, BLOCKED
- are **conceptionally** considered to be independent, concurrent control flows
- are under the control of the operating system
  - resource allocation and revocation

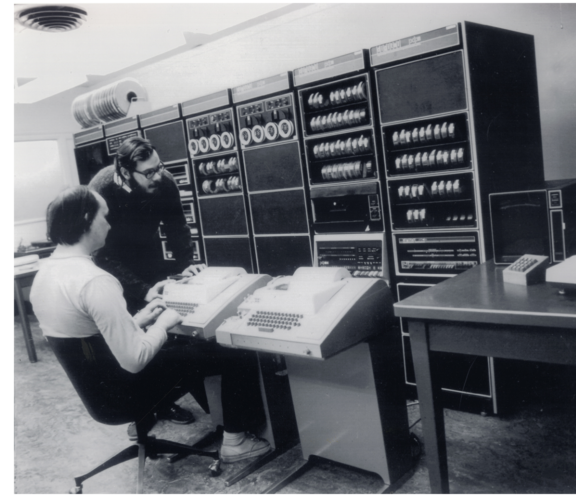




Unix (Thompson & Ritchie 1968)

Unix process model

- A system with a long (hi)story… [1,2]
- Origin: AT&T Bell Labs
  - Developed as an alternative to “Multics”
- Version 1 created on a PDP 7 [4,5]
  - written in assembler
  - 8192 18 bit words of memory
- Version 3 implemented in C on a PDP11
  - C was created to enable OS development in a high-level language

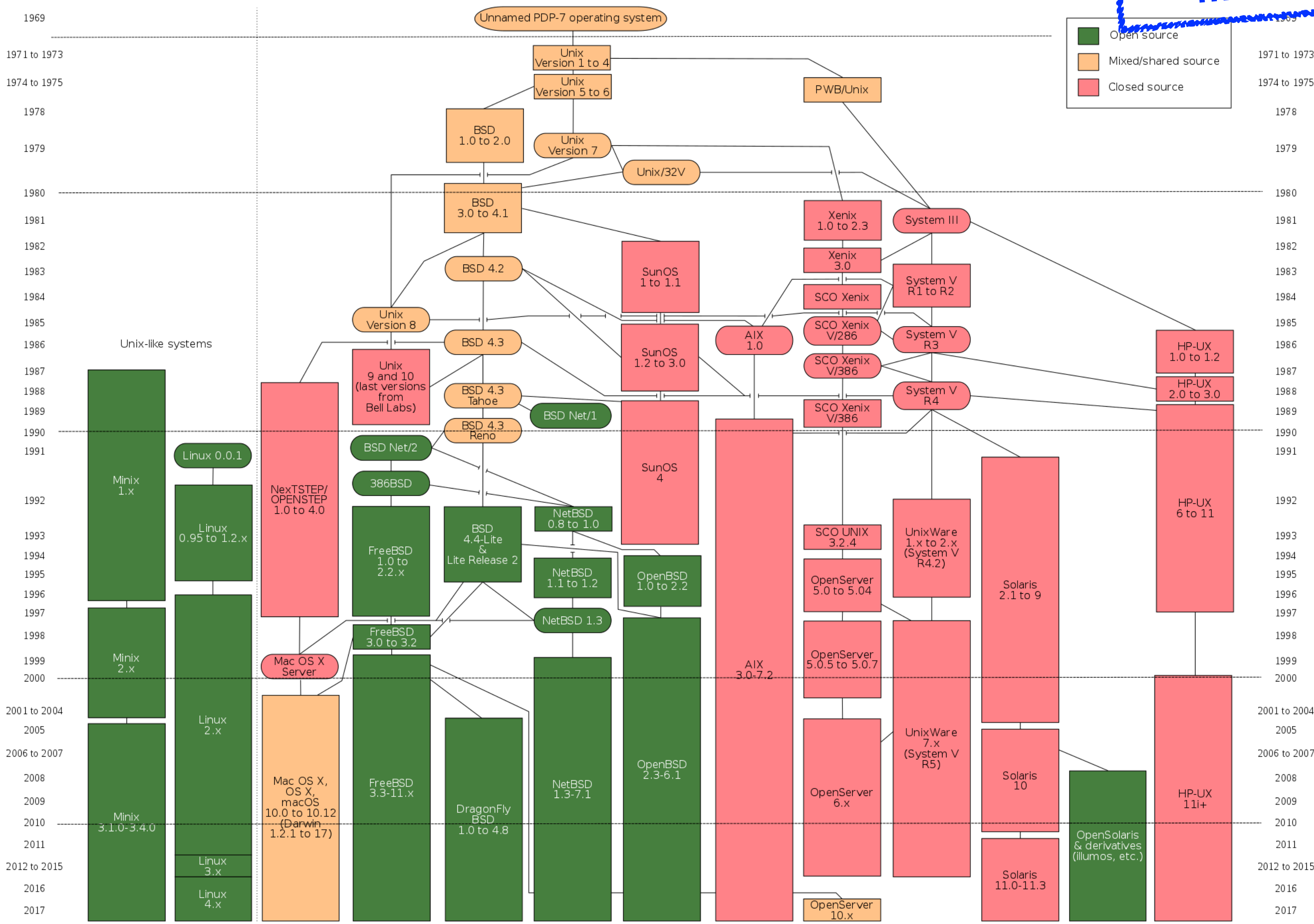


Thompson & Ritchie with a PDP11 at Bell Labs, 1970s

PDP11/40 systems in NTNU’s datamuseum

Unix variants https://www.levenez.com/unix/

Unix process model



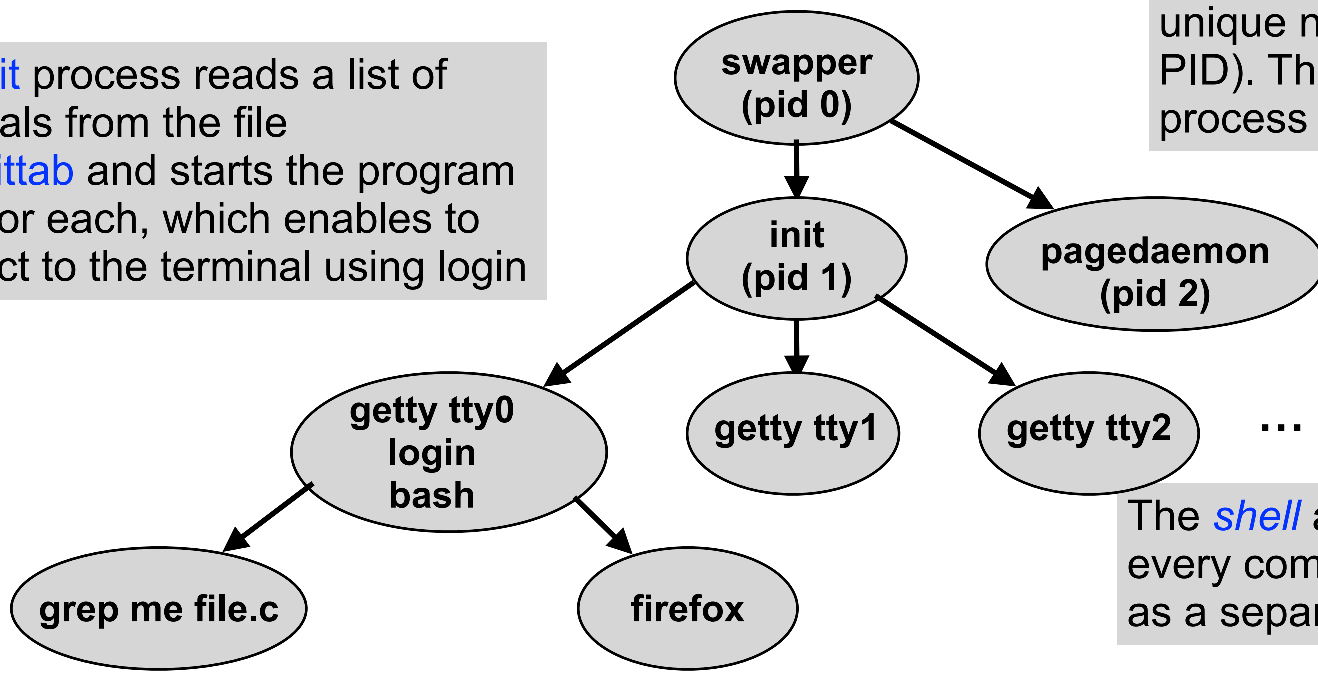
Unix processes

Unix process model

- are the primary way to **structure activities**
  - application as well as system processes
- can create new processes in a fast and easy way
  - parent process ➛ child process
- form a **process hierarchy**:

The init process reads a list of terminals from the file /etc/inittab and starts the program getty for each, which enables to connect to the terminal using login

Every Unix process has a unique number (process id, PID). The PID of the parent process is called PPID



The *shell* also uses processes: every command is executed as a separate child process

Unix *shells*

Unix process model

- A “shell” around the operating system “core”
- Text based user interface to start **commands** (Unix programs):
  - Commands can be located anywhere in the file system
  - Shell searches in directories given in the `$PATH` **environment variable**
    - e.g. /usr/bin:/bin:/usr/local/bin:...

Shell ***prompt*** your ***input***

```
me@unix:~> which vim
```

The command `which` shows where a command (here: `vim`) is found in the file system

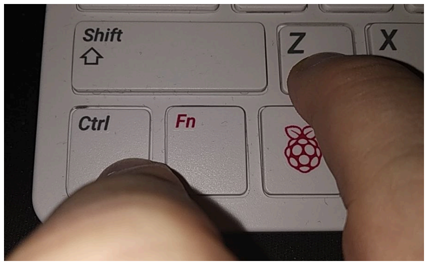
- Every executed command is a separate **child process**
- Typically, the shell blocks (waits) until the last command has terminated
- It is possible to suspend, continue and terminate commands (*job control*) and to have commands executing in the background

Unix *shells: job control*

Unix process model

```
me@unix:~> vim foo.c
```

- Command vim is started
- The shell blocks



Ctrl-Z

- Command is suspended
- The shell continues to run

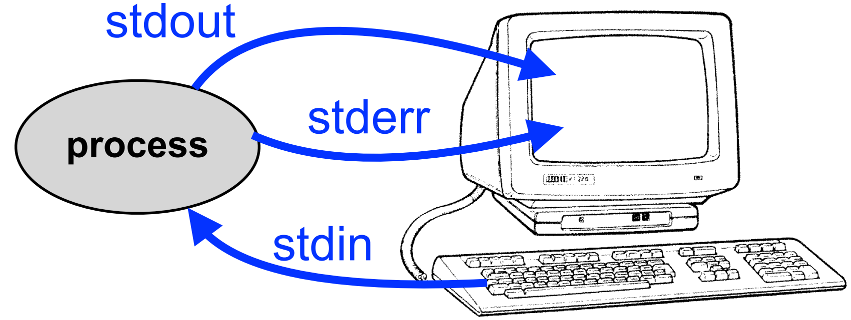
```
[1]+  Stopped    vim foo.c
me@unix:~> kate bar.c &
[2] 19504
me@unix:~> jobs
[1]+  Stopped    vim foo.c
[2]-  Running    kate bar.c &
me@unix:~> bg %1
[1]+ vim foo.c &
me@unix:~> jobs
[1]-  Running    vim foo.c &
[2]+  Running    kate bar.c &
```

- The & at the end of the input tells the shell to start and run the kate command in the background
- jobs gives a list of all started commands
- bg sends a suspended command to the background

Standard I/O channels

Unix process model

- Usually connected to the terminal in which the shell runs that started the process:
  - Standard input (stdin): read user input (Keyboard)
  - Standard output (stdout): text output of the process (terminal window)
  - Standard error (stderr): separate channel for error messages (usually also connected to the terminal)



- Almost all Unix commands also ***accept files as input or output channels*** (instead of the terminal)
- Shells provide a simple syntax to *redirect* the standard I/O channels

Redirecting standard I/O

Unix process model

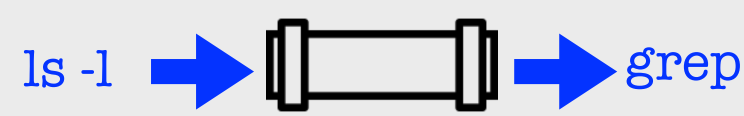
```
me@unix:~> ls -l > d1
me@unix:~> grep "Jan 29" < d1 > d2
me@unix:~> wc < d2
  2  18  118
```

Redirection of the **standard output** of the ls command to the file d1 using >

Redirection of the **standard input** of the command wc from the file d2 using <

This can also be written in a compact way:

```
me@unix:~> ls -l | grep "Jan 29" | wc
  2  18  118
```



The | **(pipe)** symbol tells the shell to connect the standard output of the left process (ls) to the standard input of the right process (grep)

The Unix philosophy

Unix process model

- Doug McIlroy, the inventor of Unix pipes, summarized the Unix philosophy as follows:

"This is the Unix philosophy:

*Write programs that **do one thing** and do it well.*

*Write programs to **work together**.*

*Write programs to **handle text streams**, because that is a universal interface."*

[by Denise Panyik-Dale, CC BY 2.0]

This is commonly expressed in a shorter way:
"Do one thing, do it well."

Process–OS interaction in Unix

Unix process model

- How does an application program request a service from the operating system?
- From the point of view of the application, calling an operating system service looks like a regular function call, e.g.:

```
pid = fork();
```

- However, arbitrarily calling code inside the OS kernel is dangerous:
  - No checking of permission to execute a function
  - No checking for correct parameters
    ➢ *security nightmare!*
- The transition from code executing in an application to code running in the kernel needs to be protected!

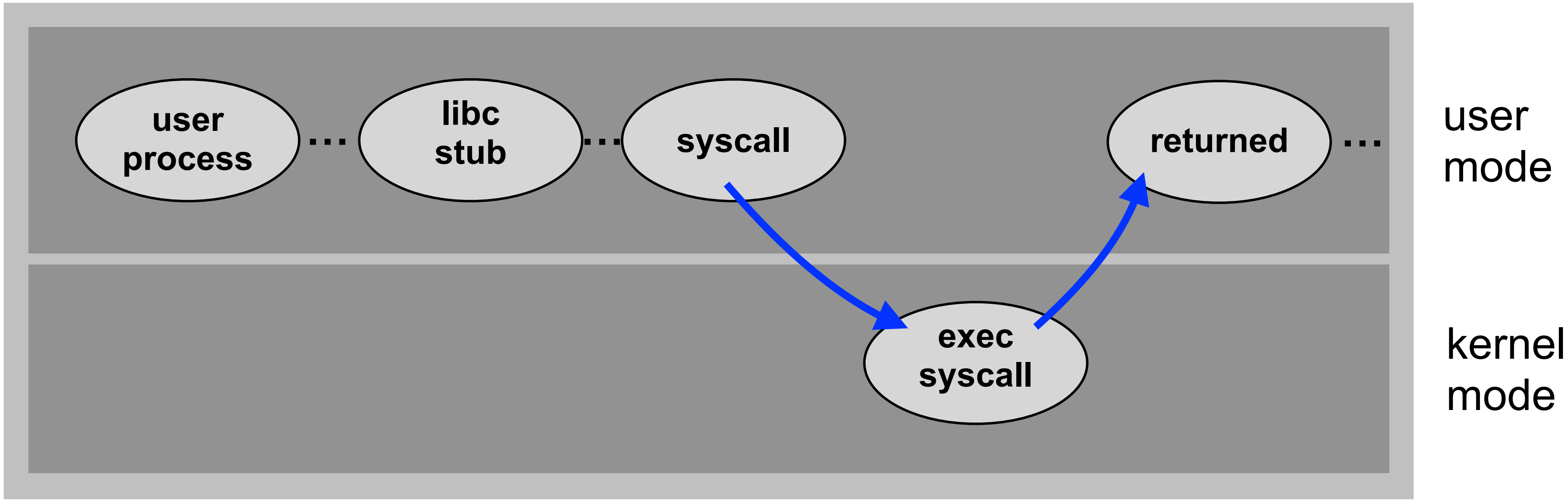
Process–OS interaction in Unix

Unix process model

- The transition from code executing in an application to code running in the kernel needs to be protected!
- Many CPUs provide several *execution modes:*
  - “user mode”: only restricted functionality is allowed
  - “kernel” or “supervisor mode”: full access to all hardware resources
- Special machine instructions are provided to transition from user to kernel mode:
  - int 0x80 (intel x86), syscall/sysenter (intel/AMD64)
  - trap (Motorola 68k), SVC (ARM), ECALL (RISC-V)
- Executing such an instruction causes the CPU to change its current execution mode to kernel mode and jump to an address predetermined by the processor hardware: ***system call***

Process–OS interaction in Unix

Unix process model



- Applications can execute a syscall instruction directly, but:
  - This stops working when the syscall interface changes
- In most modern systems, the C library (libc) provides ***stubs*** (adapter functions) that call the actuall syscall
  - The stub function is a regular function linked to the application

Unix process control: syscalls

Unix process model

- A first overview of process related system calls (syscalls) [3]

  - `getpid` (2) returns PID of the calling process
  - `getppid` (2) returns PID of the parent process(PPID)
  - `getuid` (2) returns the UID of the calling process
  - `fork` (2) creates a new child process
  - `exit` (3), `_exit` (2) terminates the calling process
  - `wait` (2) waits for the termination of a child process
  - `execve` (2) loads and starts a program in the context of the calling process

The number in brackets gives the **section** of the Unix manual pages the command is described in, read them e.g. with `man 2 wait`

Unix processes in detail: fork()

Unix process model

System call: pid_t fork (void)

This is the C prototype for fork:
no parameters (void),
returns a pid_t value

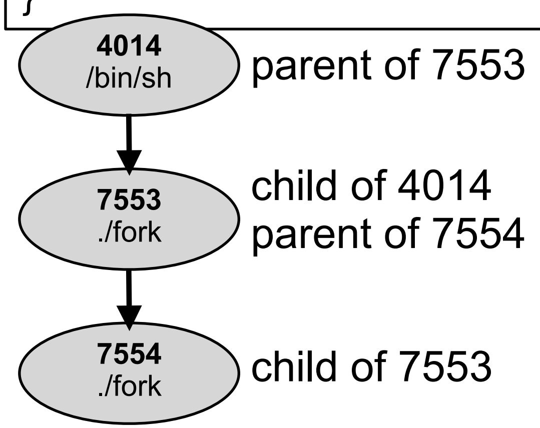
- Duplicates the calling process
  (the standard way to create new processes in Unix!)
- The child process inherits…
  - Address space (code, data, bss, stack segments)
  - User and group ID
  - Standard I/O channels
  - Process group, signal table (more on this later)
  - Open files, current working directory (also later…)
- ***Not*** copied are the following:
  - Process ID (PID), parent process ID (PPID)
  - Pending signals, accounting data, ...
- **One process calls fork, but two processes return!**

???

Use of fork()

Unix process model

```
... /* includes */
int main () {
    int pid;
    printf("In parent: pid %d PPID %d\n", getpid(), getppid());
    pid = fork();   /* Process is duplicated here!
                       Both continue running from here */
    if (pid > 0)
        printf("In the parent process, child PID %d\n", pid);
    else if (pid == 0)
        printf("In the child process, PID %d PPID %d\n",
                getpid(), getppid());
    else
        printf("Oh, an error!\n"); /* more in the theoretical ex.*/
}
```




```
me@unix:~> gcc -o fork fork.c
me@unix:~> ./fork
In parent: pid 7553 PPID 4014
In the child process,  PID 7554 PPID 7553
In the parent process, chip PID 7554
```

Discussion: fast process creation

Unix process model

- Copying the address space takes a lot of time
  - Especially if the program immediately calls exec..() afterwards ➛ *complete waste of time!*
- Historic solution: vfork
  - The parent process is suspended until the child process calls exec..() or terminates using _exit()
- The child simply uses code and data of its parent (without copying!)
  - The child process ***must not change any data***
  - sometimes not so simple: e.g., don't call exit(), but _exit()!
- Modern solution: ***copy on write***
  - Parent and child process ***share the same code and data segments*** using the memory management unit (MMU)
  - A segment is copied only if the child process changes any data
  - This is not the case when exec..() is called directly after fork()
  - fork() using copy on write is *almost* as fast as vfork()

Unix processes in detail: _exit()

Unix process model

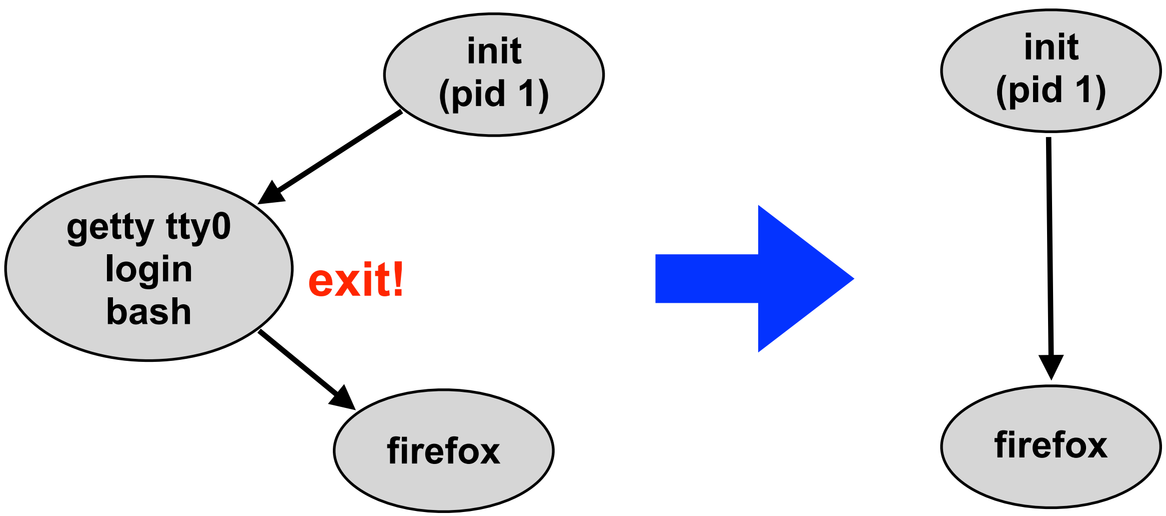
System call: void _exit (int)

- Terminates the calling process and passes the integer argument as “exit status” to the parent process
  - This call ***does not return***!
- Releases the resources allocated by the process
  - open files, used memory, …
- Sends a signal SIGCHLD to its parent process
- There is also a library function exit (3) which additionally releases resources used by the C library
  - Among other things, this outputs (*flushes*) all data still stored in output buffers!
- Normal processes should use exit (3), not _exit

Discussion: orphaned processes

Unix process model

- A Unix process is **orphaned** when its parent process terminates
- What happens to our process hierarchy?



What happens if init exits?

The **init** process (always pid 1) adopts all orphaned processes. Thus, the process hierarchy is still in working order.

Unix processes in detail: wait()

Unix process model

System call: `pid_t wait (int *)`

- Blocks the calling process until one of its child processes terminates
  - The return value of `wait` is the terminated child's PID
  - Using the `int *` parameter, the caller is passed the child's "exit status" (and more)
- `wait` returns immediately if all child processes are already terminated

Use of wait()

Unix process model

```
... /* includes, main() { … */
pid = fork();   /* create child process */
if (pid > 0) {
   int status;
   sleep(5);    /* Library function: sleep 5 seconds */
   if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit status: %d\n", WEXITSTATUS(status));
}
else if (pid == 0) {
   exit(42);
}
...
```

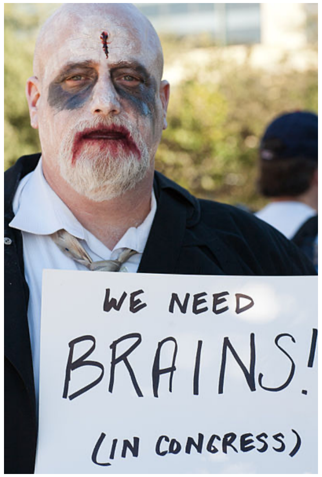
A process can also be ***"killed"*** from the outside, i.e. it does not call `exit`. In this case, `WIFEXITED` would return 0.

```
me@unix:~> ./wait
Exit status: 42
```

Discussion: *zombies*

Unix process model

- A terminated process is called a ***"zombie"*** until its exit status is requested using `wait`
- The resources allocated to such processes can be released, but the OS process management still needs to know about them
- Especially the *exit status* has to be saved




```
me@unix:~> ./wait &
me@unix:~> ps
  PID TTY          TIME CMD
 4014 pts/4    00:00:00 bash
17892 pts/4    00:00:00 wait
17895 pts/4    00:00:00 wait <defunct>
17897 pts/4    00:00:00 ps
me@unix:~> Exit status: 42
```

Example program from the previous slide during the 5 seconds waiting time

Zombies are annotated by `ps` as `<defunct>`

Unix processes in detail: execve()

Unix process model

System Call: `int execve (const char *command, const char *args[ ], const char *envp[ ])`

- Loads and starts the command passed in the “command” parameter
- Only returns in case of an error
  - e.g. command does not exist, no access, …
- Replaces the complete address space of the calling process
  - ***but it remains the same process!***
  - Same PID, PPID, open files, ...
- The C library (libc) provides some comfortable support functions that internally call `execve`: `execl`, `execv`, `execlp`, `execvp`, … (check their man pages!)

Use of exec()

Unix process model

```
... /* includes, main() { … */
char cmd[100], arg[100];
while (1) {
  printf ("Command?\n");
  scanf ("%99s %99s", cmd, arg);
  pid = fork(); /* Process is duplicated!
                   Both continue running from here. */
  if (pid > 0) {
    int status;
    if (wait(&status) == pid && WIFEXITED(status))
      printf ("Exit Status: %d\n", WEXITSTATUS(status));
  }
  else if (pid == 0) {
    execlp(cmd, cmd, arg, NULL);
    printf ("exec failed\n");
  }
...
}
```

Discussion: why no `forkexec()`?

Unix process model

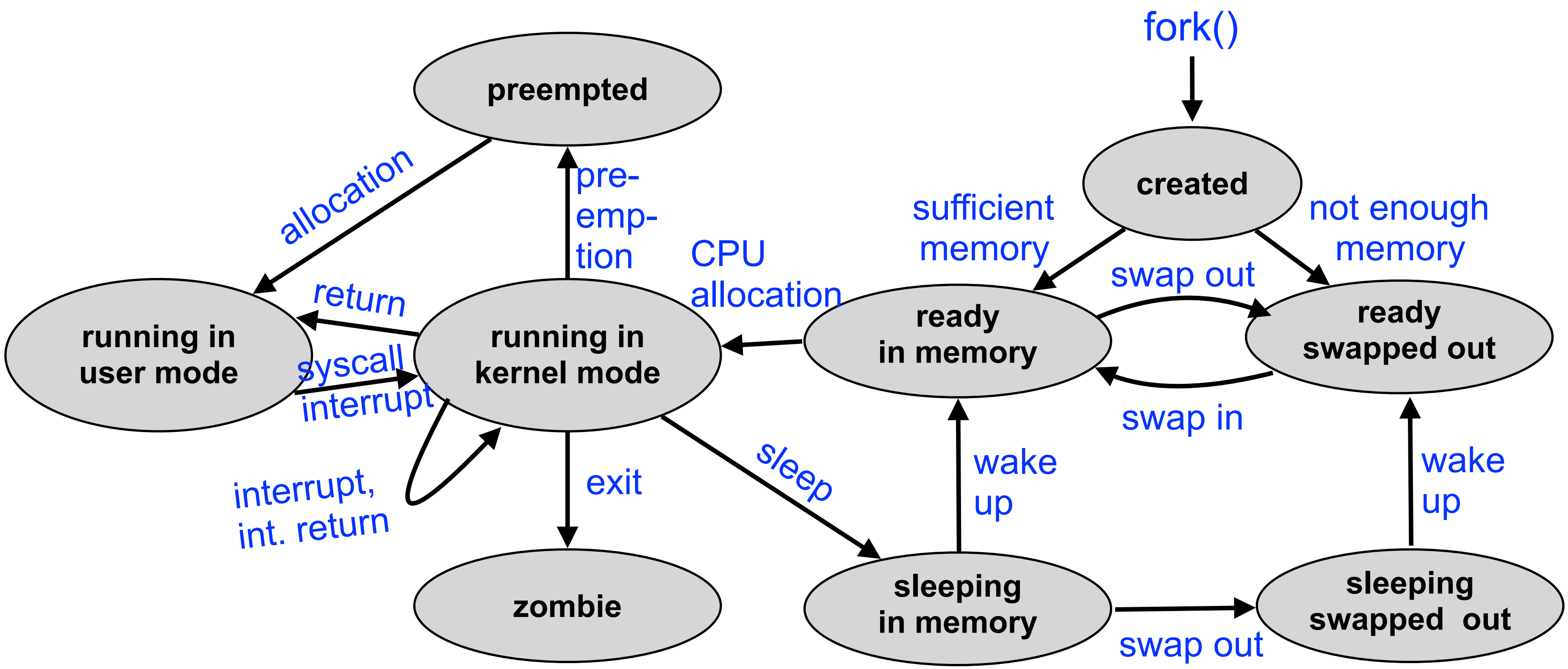
- The parent process has more control if we separate the calls to `fork` and `execve`:
  - Execute operations in the context of the child process
  - Full access to the parent processes data

- Unix shells use this feature to e.g. …
  - redirect the standard I/O channels
  - configure pipes

Unix process states

Unix process model

- a bit more complex than our earlier simple model…



Conclusion

- Process management is an important part of any OS
  - Unix has a *process hierarchy*
  - The *init* process (PID 1) is the root of the hierarchy
- Special approach taken in Unix:
  separate process creation (`fork`) and program execution (`exec`)!
  - Used by the Unix shell to implement I/O redirection
- Small set of basic system calls for process management
  - Hardware support required to make `fork` efficient
- Real-world process states are quite complex

References

1. Peter H. Salus, A Quarter Century of Unix, Addison-Wesley 1995, ISBN-13: 978-0201547771
2. Brian Kernighan, UNIX: A History and a Memoir, Independently published 2019, ISBN-13: 978-1695978553
3. W. Stevens, Stephen Rago, Advanced Programming in the UNIX Environment, 3rd Edition, Addison-Wesley 2013, ISBN-13: 978-0321637734
4. Dennis M. Ritchie and Ken Thompson. 1974. The UNIX time-sharing system. Commun. ACM 17, 7 (July 1974), 365–375. DOI:https://doi.org/10.1145/361011.361061
5. Dennis M. Ritchie, The Unix Time-Sharing System - A Retrospective, https://www.bell-labs.com/usr/dmr/www/retro.pdf