Johan J. Lukkien, j.j.lukkien@tue.nl
TU/e Informatica, System Architecture and Networking

Operating Systems 2010/2011

Processes

Johan Lukkien

Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing

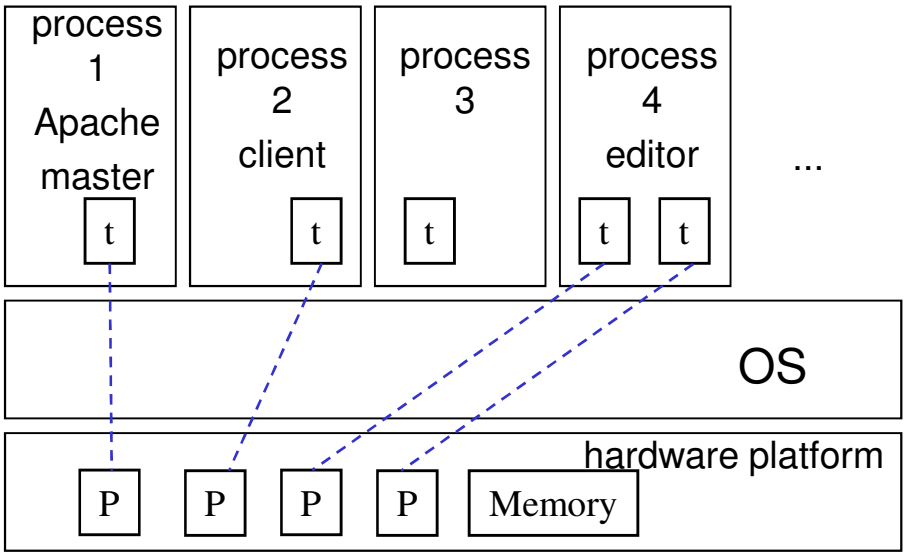
Concepts in OS: Process, Thread

- Process ("program in execution")
  - defines a data space (virtualization of the memory)
  - realizes transparency for multiple activities
  - defines ownership of resources
  - has at least one associated thread
  - unit of deployment (distribution)
  - unit of fault containment

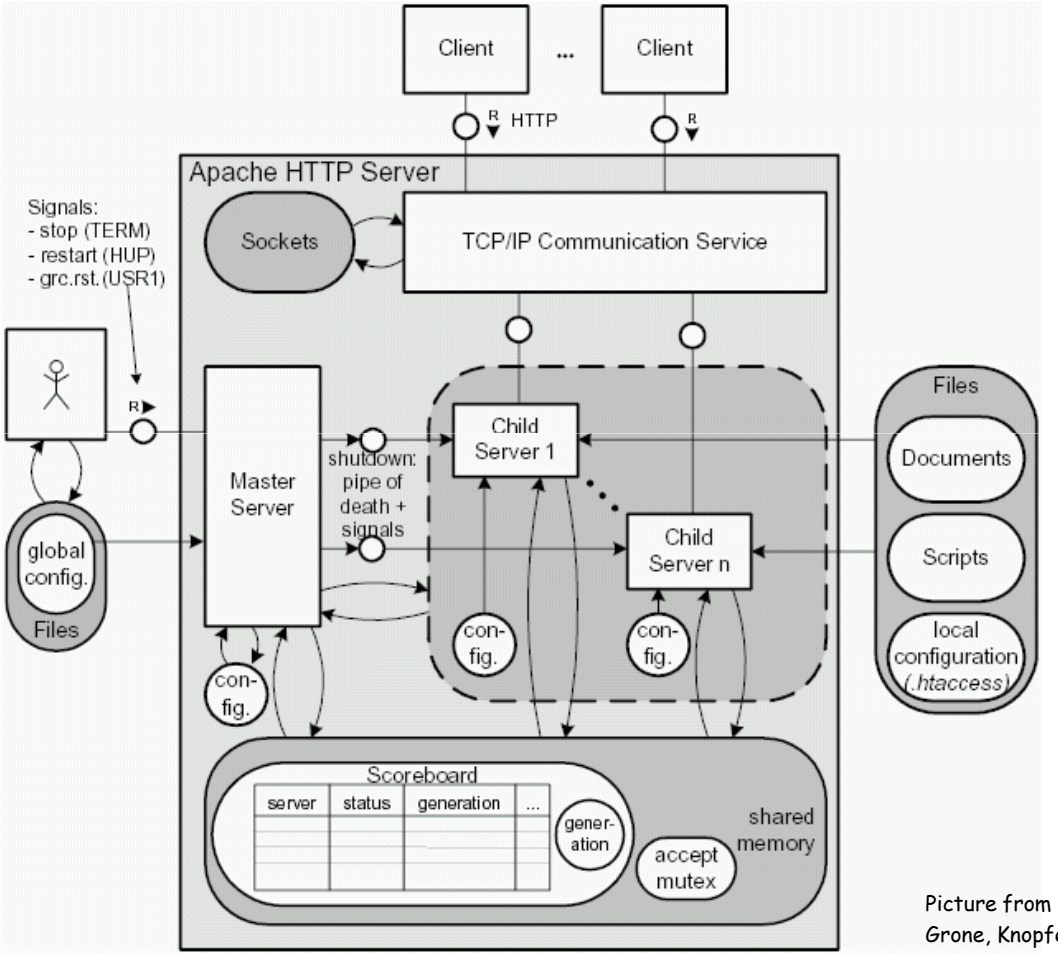
- Thread
  - unit of concurrency (virtualization of the processor)
  - unit of scheduling
    - though in case of one thread per process, the *process* is often said to be the unit of scheduling
  - operates in an address space
    - i.e., in a process; several threads may share this
  - has an associated *execution state*
    - place where it is in the code, stack image and return addresses of function calls, values of processor registers

Motivation for using processes

- Introduce safety boundaries
  - separate users
  - separate distinct, independent tasks
- Follow solution structure
  - e.g. (older) organization of the Apache web server
    - a managing process
    - client process per connection
- Increase concurrency level
- Allow for distribution



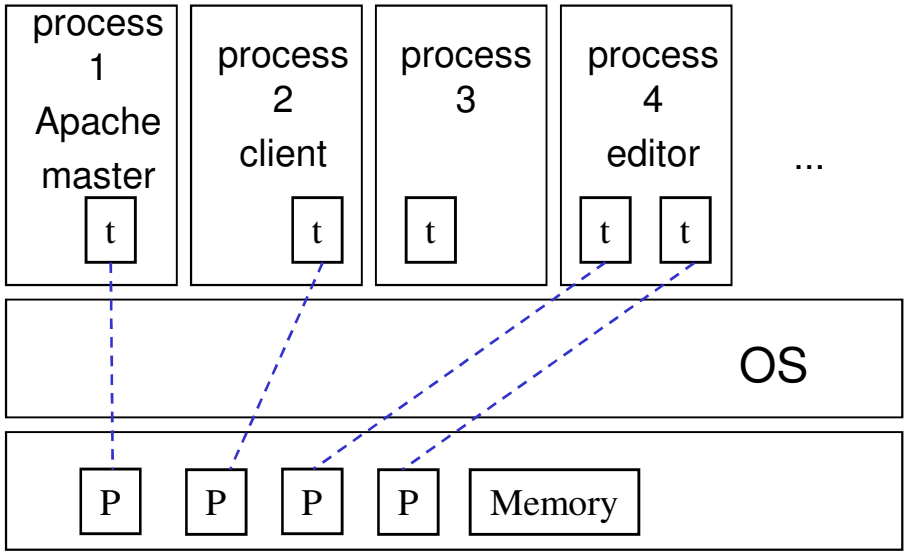
Apache organization (preforking)



Picture from 'multitasking server architecture', Grone, Knopfel, Kugel, Schmidt, Potsdam University

Reasons to introduce threads

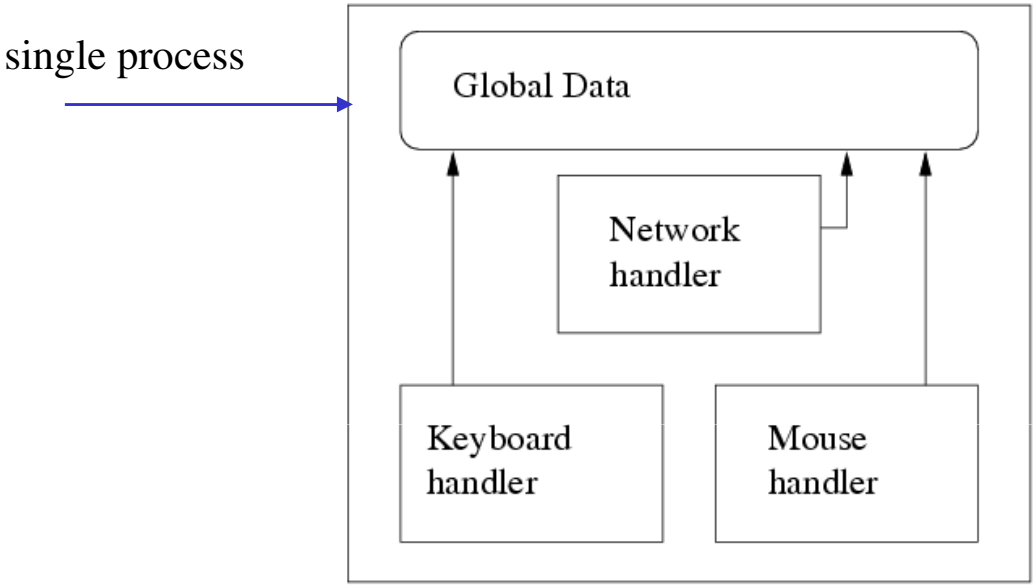
- Increase concurrency level
  - performance
    - hide latency
    - increase responsiveness
    - exploit platform concurrency
  - discriminate importance levels in activities
    - e.g. interrupt routines



- Deal with the natural concurrency
  - natural organization, structure ... e.g.
    - thread per event
    - thread per resource
    - thread per (active) external interaction sequence
      - e.g. in user interfaces

Structuring by threads

- This program must deal with three input sources



- Possible program structure:
  - while ... do
    - check keyboard;
    - check network;
    - check mouse;
    - wait a while

    od

- Simpler organization
  - start three threads
  - each thread takes care of one source

- Downside: shared resource (global data)

Two issues

1. How to *use* processes (and threads)?
   - primitives for manipulation: the API provided by the OS
     - creation, termination
     - exchange of information
       - *synchronization*: exchange events
       - *communication*: in addition, exchange data
   - interference of concurrent activities
     - what if one process modifies data another one examines?
   - program design [other courses]
     - typical parallel program structures, design criteria
       - e.g. functional-, data- and result parallelism
     - distributed systems, algorithms
2. How do we implement
   - processes, threads?
   - communication facilities?

Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing

API standardization: POSIX

- Portable Operating System Interface
  - UNIX-like
- Goal: source-code portability of applications
  - in practice: just reduce portability effort
- Standard, IEEE, ANSI, ISO, developed in API chapters:
  - mandatory ('base') & optional parts, per chapter
  - the entire set of chapters covers 'everything'
- Versioning: Posix 1003.1x-year, where x is the chapter
  - x=a, or no x: basic set of systems calls, like UNIX; 1003.1a-2004
  - x=b: real-time extensions (also: POSIX.4); 1003.1b-1993
  - x=c: multi-threading support (also: POSIX.4a); 1003.1c-1995
  - x=d: additional real-time extensions; 1003.1d-1999
  - x=j: advanced real-time extensions; 1003.1j-2000

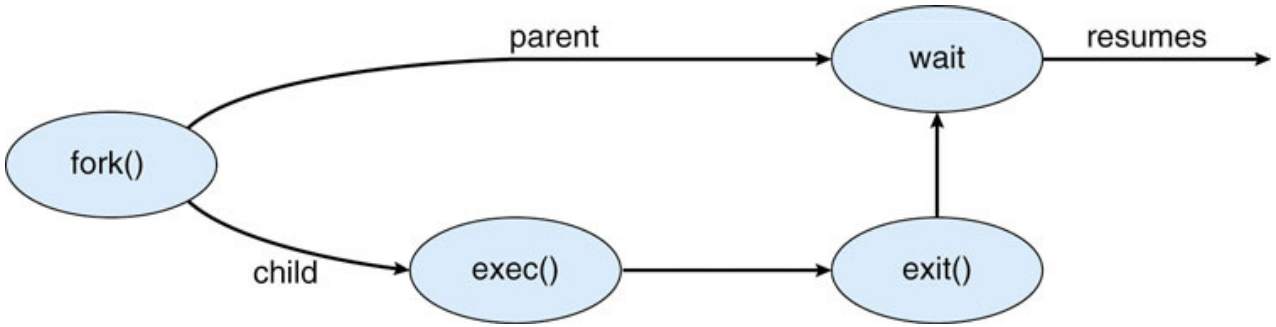
POSIX compliance

- POSIX standardizes on the API !!
  - and not on the implementation
- A system supporting POSIX provides
  - a host language and compiler (often: C)
  - interface definition files (e.g., C-header files)
    - including standardized ways to define included optional parts
  - interface implementation binary or code (e.g., C-libraries)
  - a run-time system (a platform: OS or the like)
- **Notes**:
  - POSIX is NOT any particular Unix System
  - Many OS'es have a POSIX face

POSIX processes (1003.1)

- Starting point is an already running process
  - shell (command interpreter) or other program in the GUI (windows, Xwindows)
  - that was started through a login procedure

- POSIX 1003.1 API: *fork() – exec() – exit() – wait()*



- POSIX process:
  - an executing program
  - unit of distribution, concurrency, *scheduling*
  - single address space: memory management

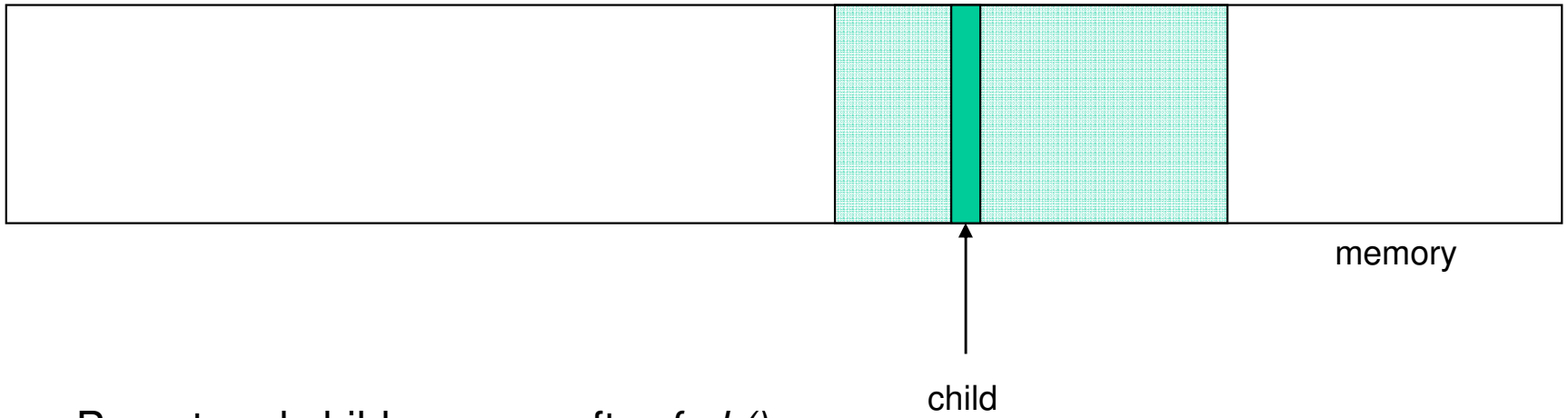
Create new process: code fragment

- *fork()* creates an identical copy of the caller in a separate memory space; only the return value stored in variable '*child*' differs between the two: 0 for the child, child-processid for the parent
- *execlp()* overwrites the calling process with its argument (file with an executable program)
- This means that code following *execlp* is never reached
- *perror()* is a generic error printing routine

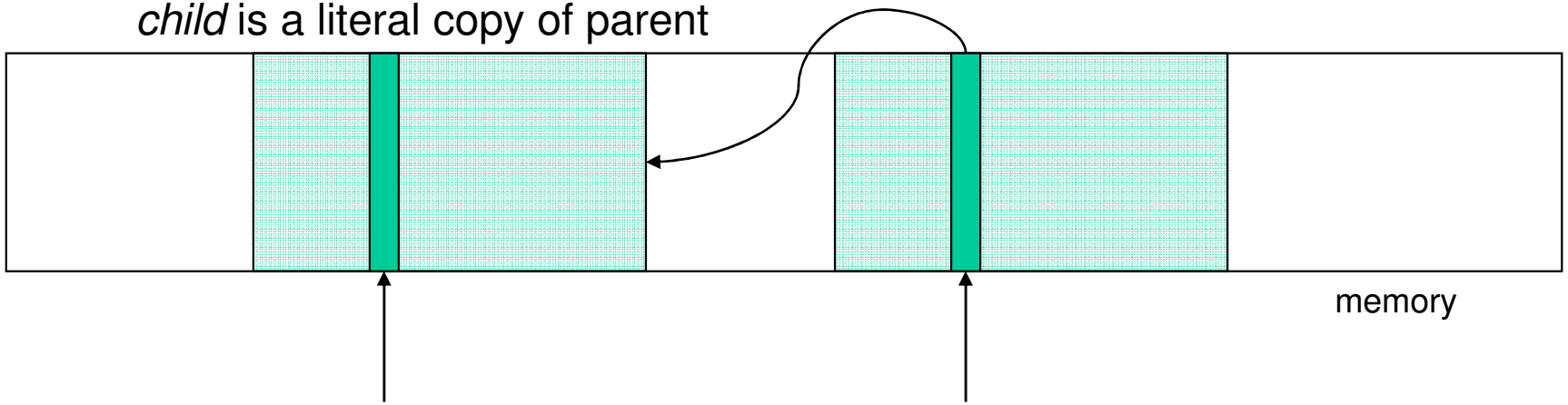
```
pid_t child;

......
child = fork();
if (child<0) /* error occurred */  { perror ("fork"); exit (-1); }

if (child == 0) /* the child */ {
      execlp ("/bin/ls", "ls", arg0, arg1, ..., NULL);
      /* this place is reached only in case of error */
      perror ("execlp"); exit (-1);
}
else /* the parent; child == process id of child */ {
      /* do whatever you want, e.g., just return from this routine */
}
......
```

Before and after a *fork()*

Parent process about to perform *child = fork()*



Parent and child process after *fork()*
child is a literal copy of parent



variable *child* of child process equals 0

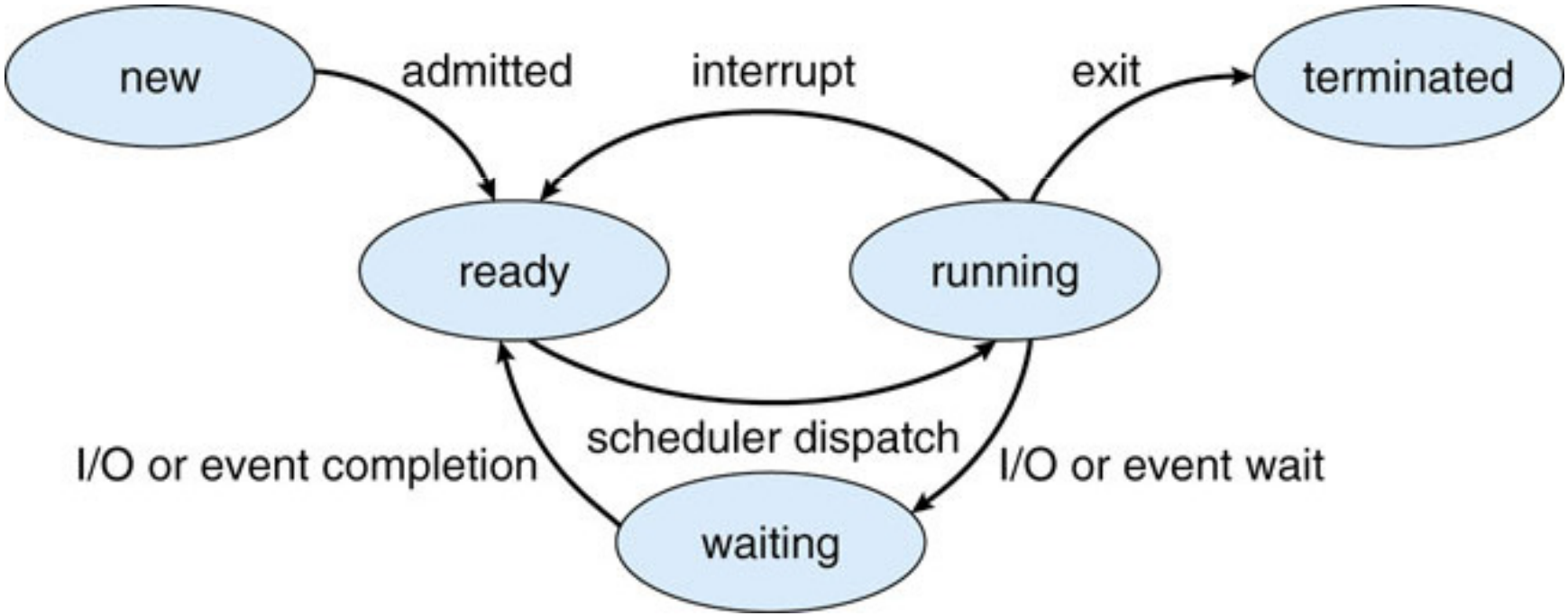
variable *child* in parent points to child process

Termination of children: fragment

- Use *exit(status)* to terminate
- Need to wait for children to free child's resources
  - on many systems at least
- Functions *wait(), waitpid()*
  - *wait()* blocks until a child returns
- Alternative: (not shown here): use asynchronous notification: signals

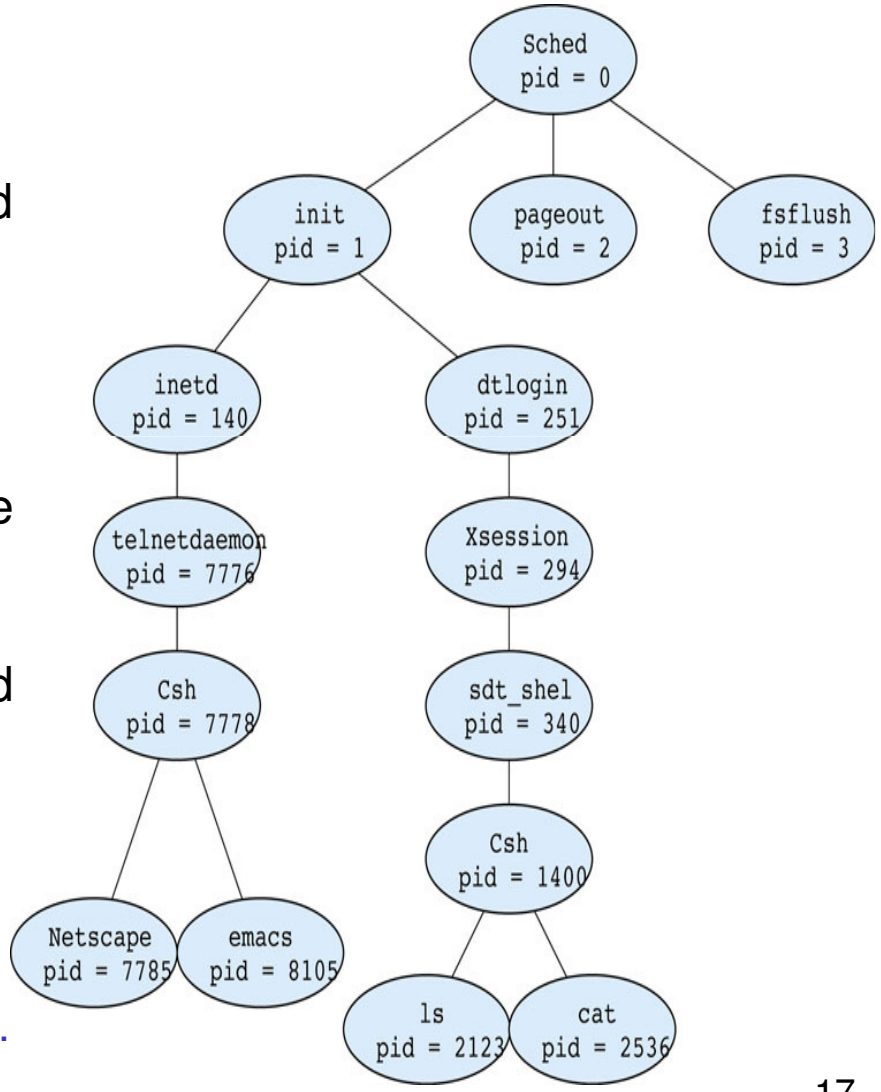
```
child:
      ..... exit (23);

parent:
      pid_t child, terminated; int status;

      /* blocking wait */
      while (child != wait (&status)) /* nothing */;

      /* or polling wait */
      terminated = (pid_t) 0;
      while (terminated != child) {
          terminated = waitpid (child, &status, WNOHANG);
          /* other useful activities */
      }
      /* both cases: status == 23 */
```

The life of a process



POSIX/UNIX processes in operation

- Processes receive a unique number through which they can be referred
- Parent-child relationships are recorded
- Orphants are attached to the system 'grandfather' called *init* (process # 1)
- *Init* is also the process that initiates the login of new users
- System services are processes started by *init*, by themselves capable of starting new ones
- Network services are executed under control of *inetd* (the internet daemon)
  - e.g. RPC support, ftp, telnet, rlogin etc.



Some processes on the SANDPIT cluster

```
/home/johanl> ps uax
USER       PID %CPU %MEM    VSZ   RSS TTY      STAT START   TIME COMMAND
root         1  0.0  0.0   1344   424 ?        S    Feb14   0:26 init [3]   --in]
root         4  0.0  0.0      0     0 ?        S    Feb14   0:03 [keventd]
root         7  0.0  0.0      0     0 ?        S    Feb14   1:58 [kswapd]
root         8  0.0  0.0      0     0 ?        S    Feb14   0:37 [bdflush]
root         9  0.0  0.0      0     0 ?        S    Feb14   1:07 [kupdated]
root        10  0.0  0.0      0     0 ?        S    Feb14   0:00 [mdrecoveryd]
root        11  0.0  0.0      0     0 ?        S    Feb14   0:00 [kcopyd]
root        12  0.0  0.0      0     0 ?        S    Feb14   0:00 [kmirrord]
root       190  0.0  0.0      0     0 ?        S    Feb14   0:00 [khubd]
root      8554  0.0  0.0   1700   760 ?        Ss   Feb14   0:00 /sbin/devfsd /d
root      8849  0.0  0.0      0     0 ?        S    Feb14   3:20 [kjournald]
root      9874  0.0  0.0   1688   648 ?        Ss   Feb14   1:36 /usr/sbin/syslo
root     10000  0.0  0.0   1480   520 ?        Ss   Feb14   0:00 /sbin/dhcpcd -H
named    10129  0.0  0.0  11868  1232 ?        S    Feb14   0:02 /usr/sbin/named
root     10214  0.0  0.0   3172   860 ?        Ss   Feb14   0:03 /usr/sbin/sshd
rpc      10350  0.0  0.0   1576   620 ?        Ss   Feb14   1:09 /sbin/portmap
root     10498  0.0  0.0   3944   624 ?        S    Feb14   3:37 /usr/sbin/ypser
root     10512  0.0  0.0   5848   680 ?        S    Feb14   0:25 /usr/sbin/ypbin
nobody   10569  0.0  0.0   1592   632 ?        Ss   Feb14   0:00 /sbin/rpc.statd
root     10579  0.0  0.0      0     0 ?        S    Feb14   3:13 [nfsd]
root     10580  0.0  0.0      0     0 ?        S    Feb14   0:00 [lockd]
root     10581  0.0  0.0      0     0 ?        S    Feb14   0:06 [rpciod]
root     10582  0.0  0.0      0     0 ?        S    Feb14   2:48 [nfsd]
root     10590  0.0  0.0   1548   592 ?        Ss   Feb14   0:28 /usr/sbin/rpc.m
root     11266  0.0  0.0   1976   624 ?        Ss   Feb14   0:03 /usr/sbin/xinetd
    -pidfile /var/run/xinetd.pid -stayalive -reuse
johanl   16966  0.0  0.0   2872  1804 pts/12   Ss   14:45   0:00 -tcsh
johanl   17221  0.0  0.0   2348   848 pts/12   R+   15:06   0:00 ps uax
```

Simple command interpreter

```
int
main(void)
{
    char    buf[MAXLINE];
    pid_t   pid;
    int     status;

    printf("%% ");    /* print prompt (printf requires %% to print %) */
    while (fgets(buf, MAXLINE, stdin) != NULL) {                (1)
        buf[strlen(buf) - 1] = 0;    /* replace newline with null */
                                                                (2)
        if ( (pid = fork()) < 0)                                (3)
            err_sys("fork error");

        else if (pid == 0) {         /* child */
            execlp(buf, buf, (char *) 0);                       (4)
            err_ret("couldn't execute: %s", buf);
            exit(127);
        }

        /* parent */
        if ( (pid = waitpid(pid, &status, 0)) < 0)              (5)
            err_sys("waitpid error");
        printf("%% ");
    }
    exit(0);
}
```

Explanation

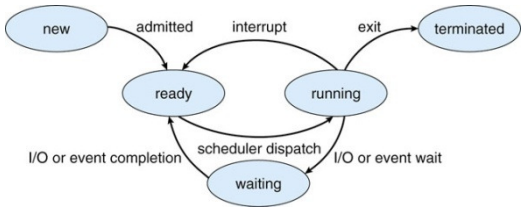
1. **fgets** reads one line at a time.
   - It returns a null pointer for a single EOF (end-of-file) character.
2. **strlen** gives the length of a string (excluding the last \0).
3. **fork** creates an exact copy of the old process.
   - It returns 0 to the child (= new) process.
   - It returns the process ID of the child process to the parent (= old) process.
4. In the child process **execlp** replaces the process by a new program.
5. The parent calls **waitpid** to wait for the child (**pid**) to terminate.
   - Otherwise the two processes run concurrently.
   - **waitpid** also returns the termination status of the child process; in the example this is ignored.
6. Main limitation of this example:
   - Commands with arguments are not possible.

Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing

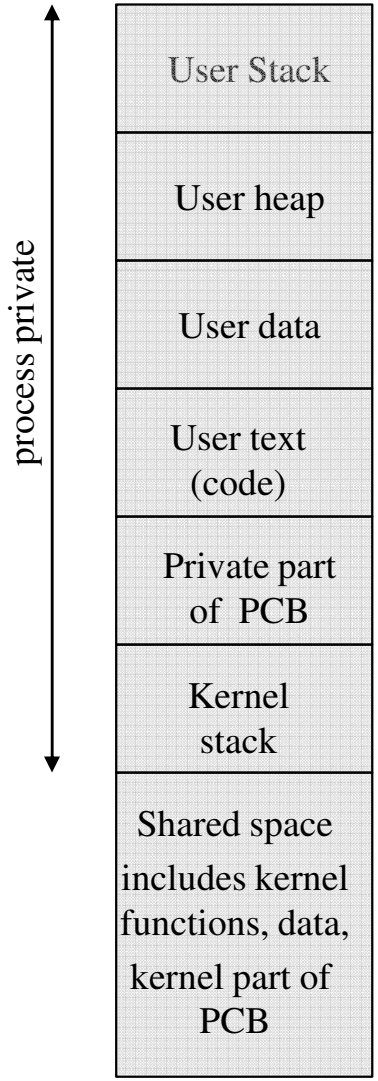
Process State

- A process is represented by a *Process Control Block* (PCB) "inside" the kernel
  - all info needed to run it
  - identified by an ID or a pointer to this block
    - dependent on representation details, e.g. table, linked list
    - we use: *self* to refer to the current process PCB
  - *state:* ready, waiting, ...
    - + perhaps the queue where the process is on
  - *scheduling queues*
    - queues where this process is part of
  - "..." includes
    - quota limits
    - accounting information
      - e.g. times in user, kernel mode
    - user identification
    - permissions

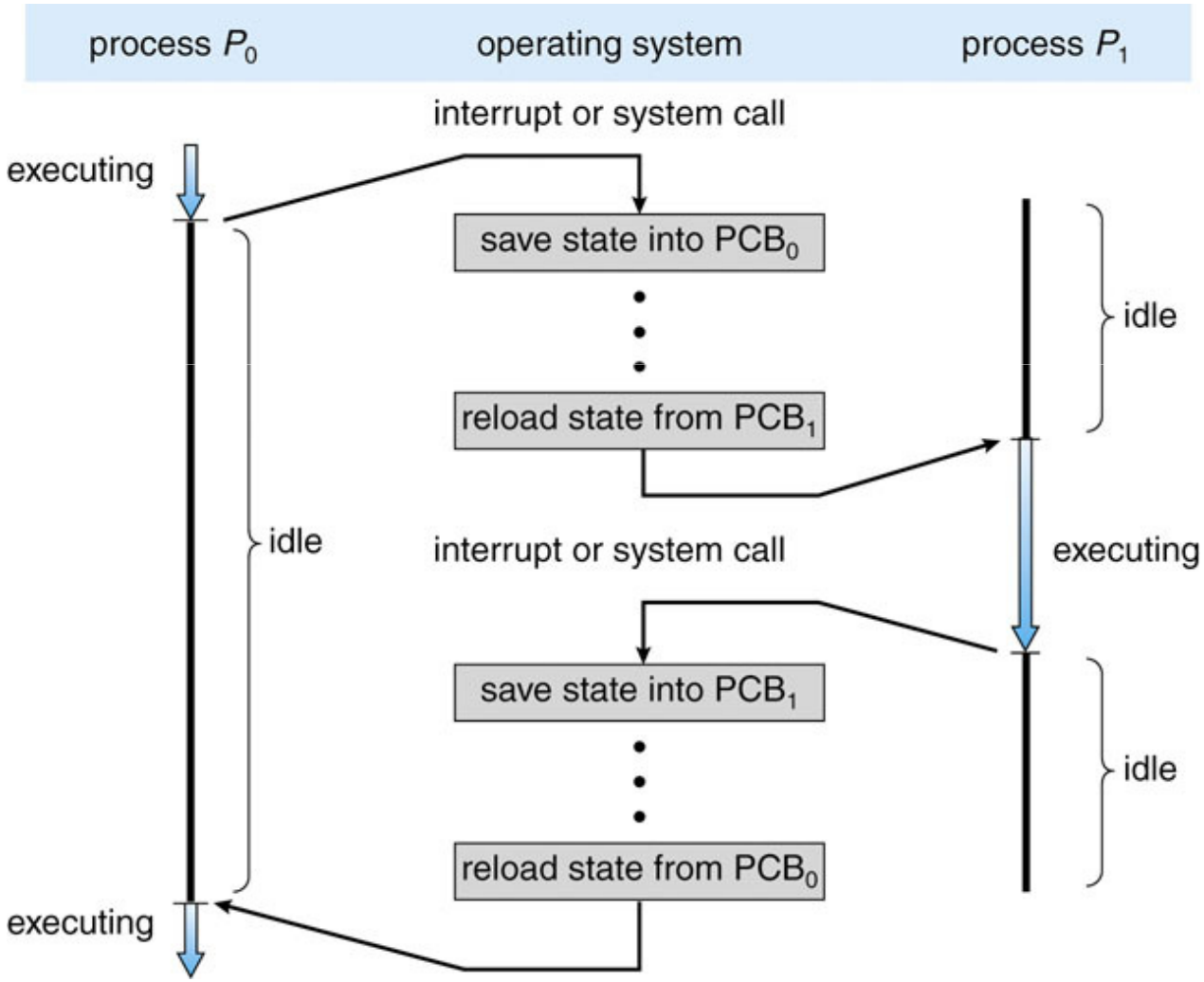


Typical memory layout for a process

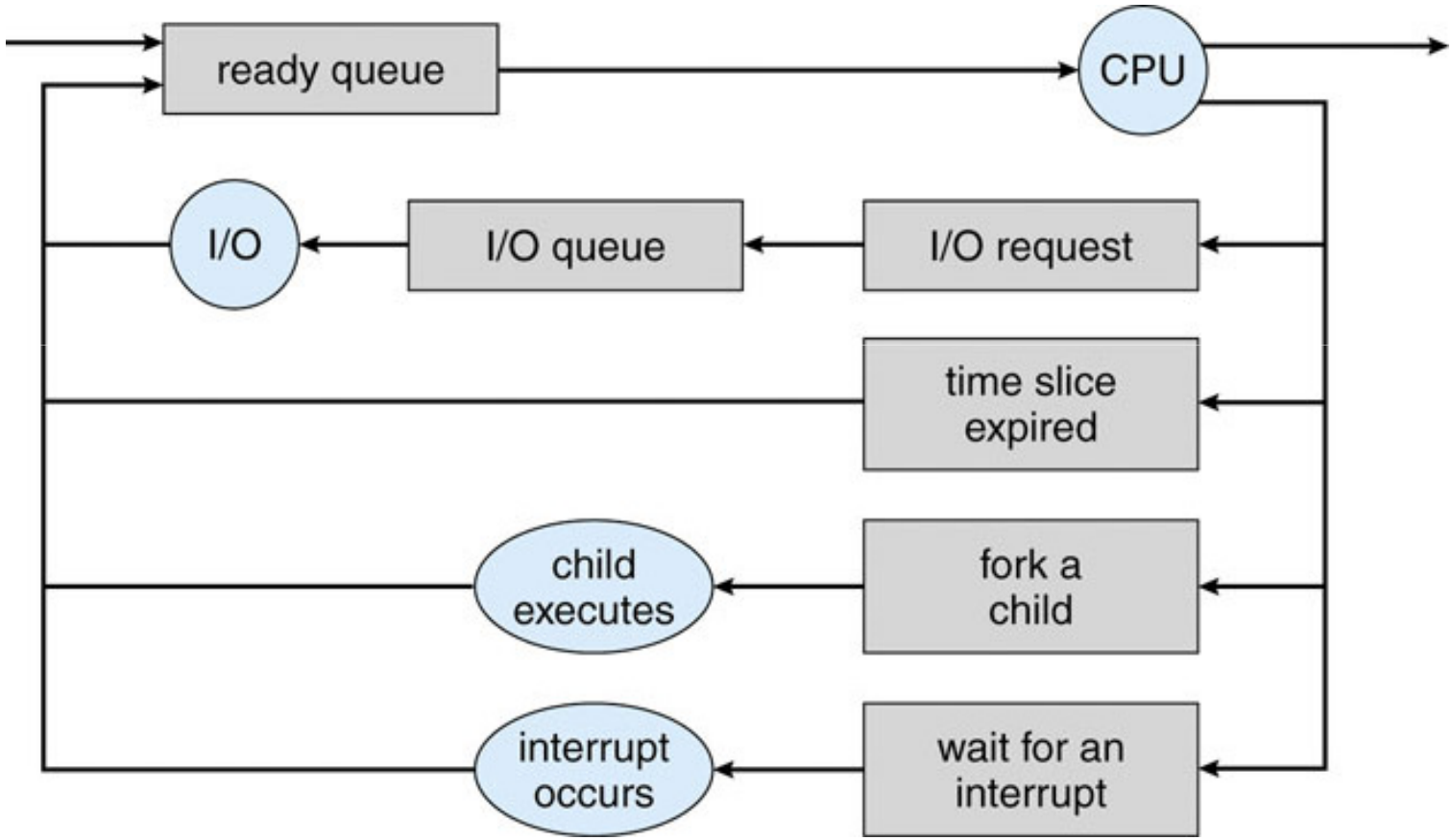
- In user mode, the process can access just its private data and stack
- In kernel mode, the process can access kernel functions and data
  - it uses a new stack for that purpose, stack pointer loaded as part of the trap
  - the kernel runs, in fact, on behalf of the process
    - "the process runs in kernel mode"
  - this space is shared by all processes
- No other processes are visible
  - hence, in order to deliver data to another process, this data must be copied in between
  - the kernel cannot deliver data to just any process; it first must change the memory settings such that this process becomes visible
- Many system calls are just a trap



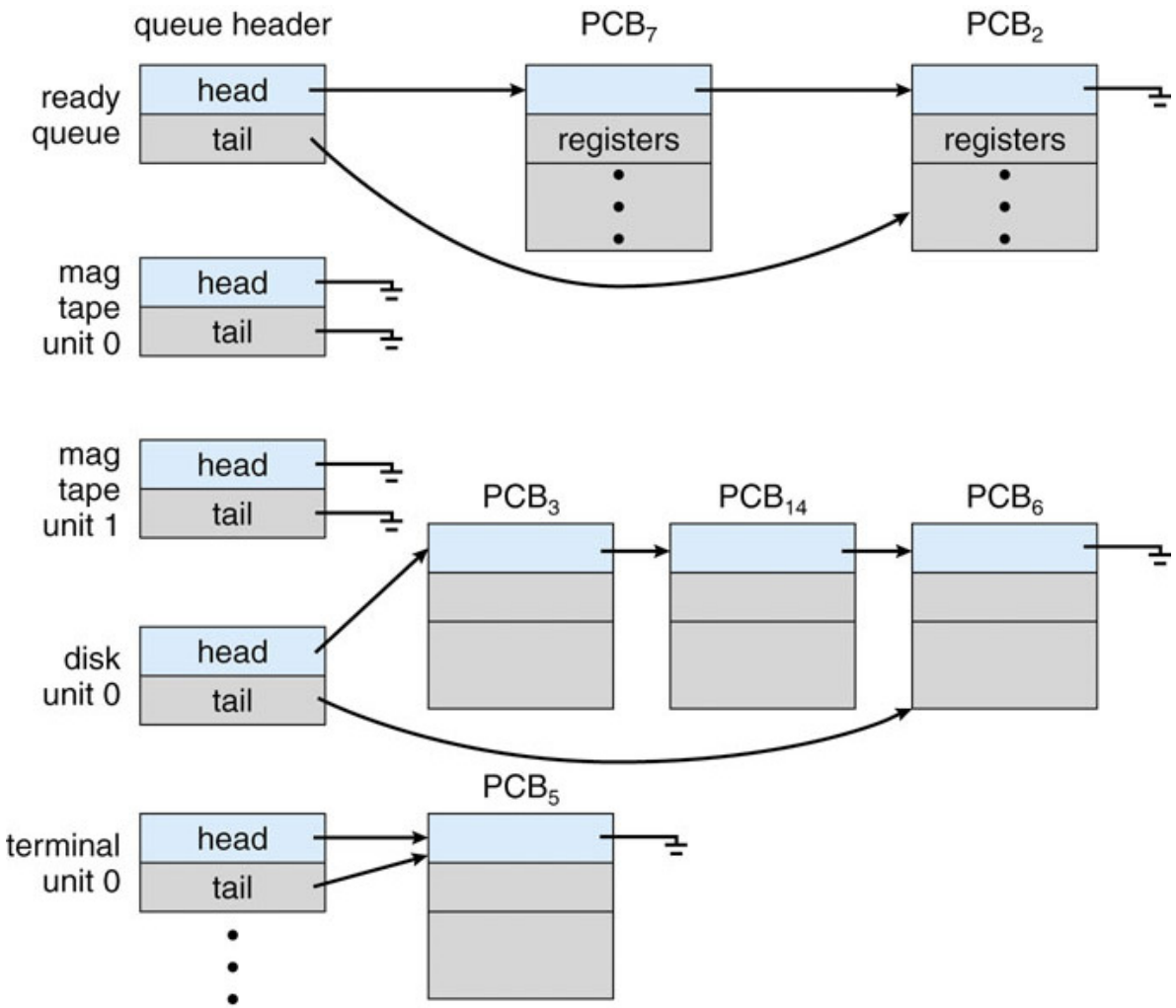
Context switch



Blocking events, queues



Link PCB at place of blocking

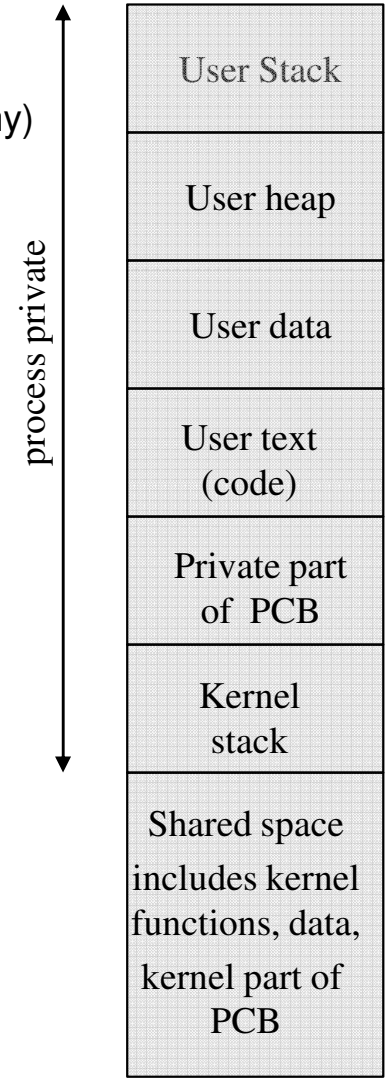


Example: process creation

- Executed
  - in response to an application call (e.g. *fork()*)
- Create new PCB, fresh process number, ...
- Fill in:
  - parent: *self* (caller)
  - initial CPU and memory state vector (defaults)
  - other control fields based on defaults and parent
- Add to ready queue
- Return status (process id) to caller
  - i.e., put this at the right place in memory
- Transfer control to scheduler
  - may need to reschedule, depending on priority
    - otherwise, just return, only mode switch needed

Example: process deletion

- Executed
  - in response to a call
    - from process itself *(exit()*) or from other one (kernel, or higher in hierarchy)
- Delete all descendents
  - may need to stop several processors!
- Return all resources
  - close open files
    - references to administration in other kernel datastructures
  - reclaim memory resources
  - *if parent waiting*, move parent to ready queue
    otherwise store status and move to 'zombie' queue
  - notice: *resources referenced through the private space of a process must be returned by the process itself (or: within the context of that process)*
    - we come back to this point later
- Transfer control to scheduler
  - the current process may have been removed
  - waiting processes (parent) may have become unblocked

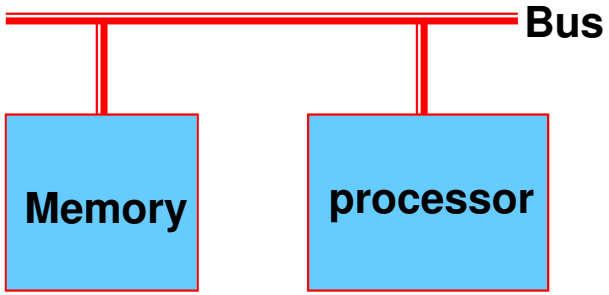


Agenda

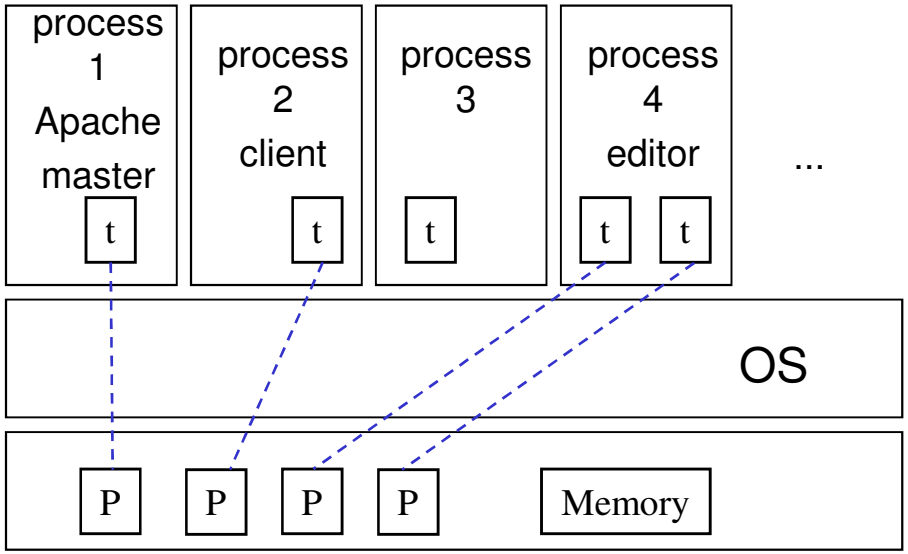
- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing

Concurrent systems

- components in a PC



- processes in a multi-tasking environment



- threads within a process

Major issues in concurrency

- **co-operation**:
  - sharing resources (hardware like printer, scanner, disk, ..., or most likely: memory and processors)
  - transfer information: *synchronization* and *communication* (needs shared resources)

- **interference**, mutual influence (good or bad):

  Assumptions or knowledge that one process has about the state, are disturbed by actions of another process

  - *good interference*: wait for another proces to set a boolean to *true* indicating delivery of a value in a variable
    - *x := false*; *{ ¬x }* **while** ¬*x* **do** *skip* **od**; "use *y*" || .... *y := E*; *x := true* .....
    - Question: are there 'tricky' interleavings?

  - *bad*: access a resource (e.g. a printer) after checking its availability; in between the check and the use the resource is accessed by another process
    - **if** *avail* **then** *{ avail }* *avail := false; "use resource"; avail := true;* **fi** || ...same

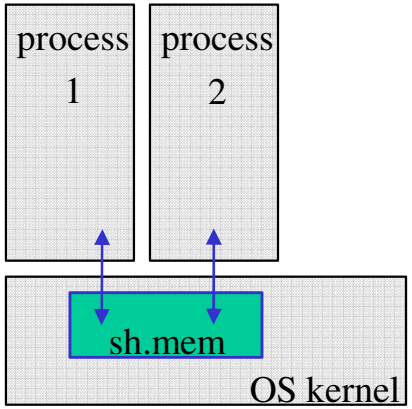
Looking forward: communication & synchronization primitives

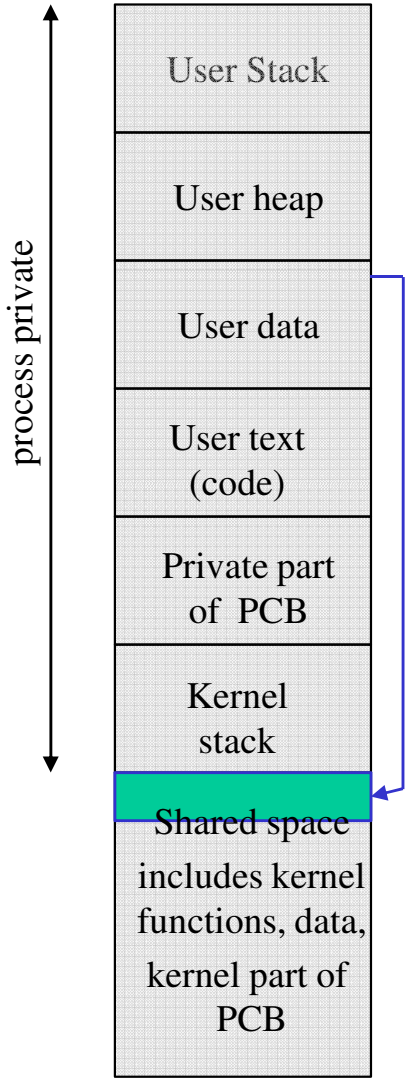
- Synchronization
  - just shared variables, e.g., event flags
    - in combination with *busy waiting,* see previous examples
  - semaphores, mutexes
  - condition variables
  - readers/writers locks
  - monitors

- Communication
  - shared memory
  - message passing (asynchronous, buffered or synchronous)
    - streaming: pipes, fifos, sockets
    - structured: message queues
    - has *implicit* synchronization (why?)
  - remote procedure calls (combines functionality and messaging)
  - signals
    - asynchronous messages, i.e., interrupt the flow of control
    - may regard the *signal handler as* a concurrent thread, that is (synchronously) waiting for an event

Shared memory between processes

- A memory segment is reserved in the kernel memory
- It is added to the space addressable in user mode
- Since it is in the shared space, it is available to other processes
- *No kernel involvement is required upon subsequent reading and writing*





POSIX: Shared Memory API (1003.1)

int shmget(key_t *key*,size_t *size*, int *shmflg*);

/* key: an identifier, for reference
 * size: in #bytes
 * shmflg: indicates how to open, e.g. readonly, create if does not exist
 * shmget() returns an id for manipulation
 */

void *shmat(int *shmid*, const void **shmaddr*, int *shmflg*);

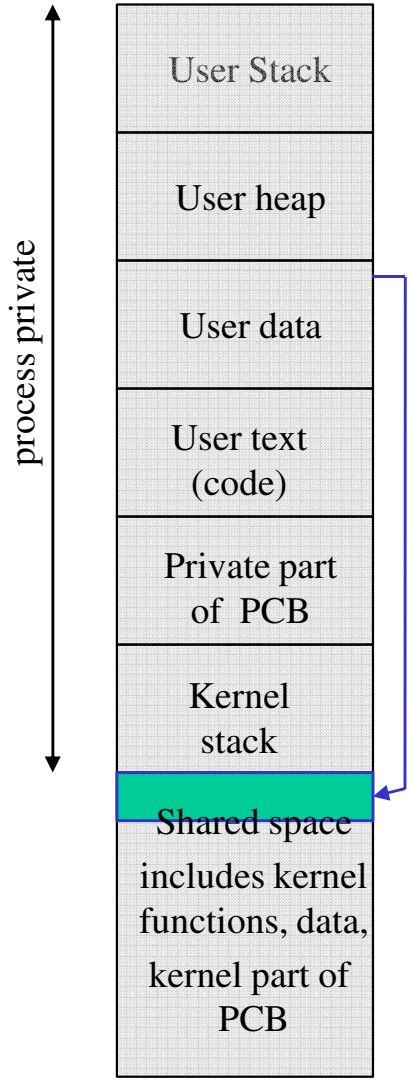
/* shmid: from shmget()
 * shmaddr: place in process space where shared mem should be referenced
 * shmflg: detail and steer the operation
 * shmat() returns a pointer to a shared byte array of size bytes (see shmget())
 */

int shmdt(const void *shmaddr);

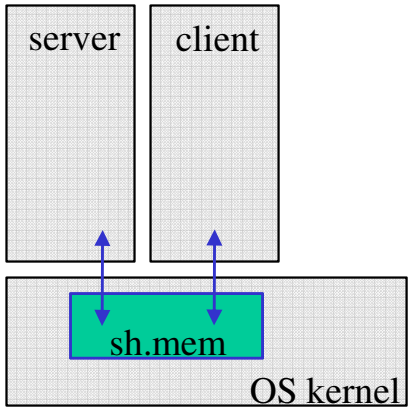
/* shmaddr: shared memory to detach, obtained from shmat()
 * shmdt returns just error condition
 */

int shmctl(int *shmid*, int *cmd*, struct shmid_ds **buf*);

/* control operations, cmd = IPC_RMID removes the shared memory */



Example



- from Dave Marshall, http://www.cs.cf.ac.uk/Dave/C/node27.html
- Server
  - creates shared memory
  - writes ‘ab...z’ into it
  - (busy) waits until ‘a’ is replaced by ‘*’
- Client
  - attaches to (not: creates) the created shared memory segment
  - reads and prints the content
  - puts a ‘*’ at the expected place

Server

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    char c;
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* arbitrary name */
    /* Create the segment */
    if ((shmid = shmget(key, SHMSZ,
                IPC_CREAT | 0666)) < 0) {
        perror("shmget"); exit(1);
    }

    /* Now we attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* Now put some things into the memory for the
     * other process to read.
     */
    s = shm;
    for (c = 'a'; c <= 'z'; c++) *s++ = c;
    *s = NULL;

    /* wait until the other process changes the first
     * character of our memory to '*'
     */
    while (*shm != '*') /* sleep(1) */;

    exit(0);
}
```

Client

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <stdio.h>
#define SHMSZ     27

main()
{
    int shmid;
    key_t key;
    char *shm, *s;

    key = 5678; /* chosen in server */

    /* Locate the segment */
    if ((shmid = shmget(key,
            SHMSZ, 0666)) < 0) {
        perror("shmget"); exit(1);
    }

    /* attach the segment to our data space */
    if ((shm = shmat(shmid, NULL, 0)) == (char *) -1) {
        perror("shmat"); exit(1);
    }

    /* Now read what the server put in the memory. */
    for (s = shm; *s != NULL; s++) putchar(*s);
    putchar('\n');

    /* Change the first character of the segment to '*' */
    *shm = '*';

    exit(0);
}
```

Output

```
user@ubuntu8041:~/2IN05$ ./SHMserver &
[3] 19817
user@ubuntu8041:~/2IN05$ ./SHMclient
abcdefghijklmnopqrstuvwxyz
user@ubuntu8041:~/2IN05$ ./SHMclient
*bcdefghijklmnopqrstuvwxyz
[3]-  Done                    ./SHMserver
user@ubuntu8041:~/2IN05$ ./SHMclient
*bcdefghijklmnopqrstuvwxyz
user@ubuntu8041:~/2IN05$
```

- Notes:
  - busy waiting (in this example as well as in the book) is not an acceptable synchronization method at the level of the OS
    - it uses the entire timeslice and destroys performance
    - the *sleep()* avoids continuous checking at the expense of a one second delay
  - instead, synchronization methods provided by the OS should be used
    - message passing, or one of the methods discussed later
  - since the shared memory is not removed it remains after termination of server and client
    - to remove: add *shmctl (shmid, IPC_RMID, NULL)*
  - there is a newer version of these primitives that should be used

Agenda

- Processes, threads
- API for creation and termination
  - POSIX
  - Processes
- Implementation aspects
- Communication and Synchronization
  - Shared Memory
  - Message passing

POSIX: pipes & Fifo's (1003.1)

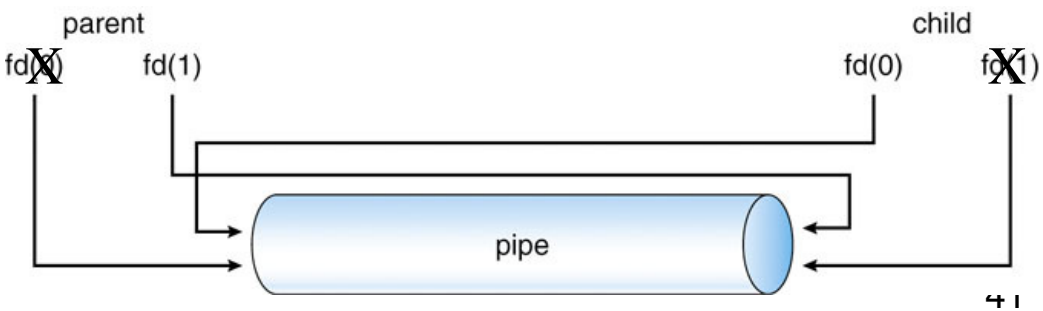
- Pipe:
  - connected pair of file(-descriptor)s
  - buffered, one-directional *streams* between parent and child
  - process
    - no message structure imposed

- Fifo ("named pipe"):
  - same as a pipe, but connection setup via name in kernel
  - create a unique name, e.g., "/tmp/fifo", using
    - *mkfifo ("/tmp/fifo", mode);*
  - Now use "/tmp/fifo" as a file: *open(), read(), write()*
  - Admits multiple writers
    - check the manual for the rules on atomicity of write operations

Pipes & Fifo's (cnt'd)

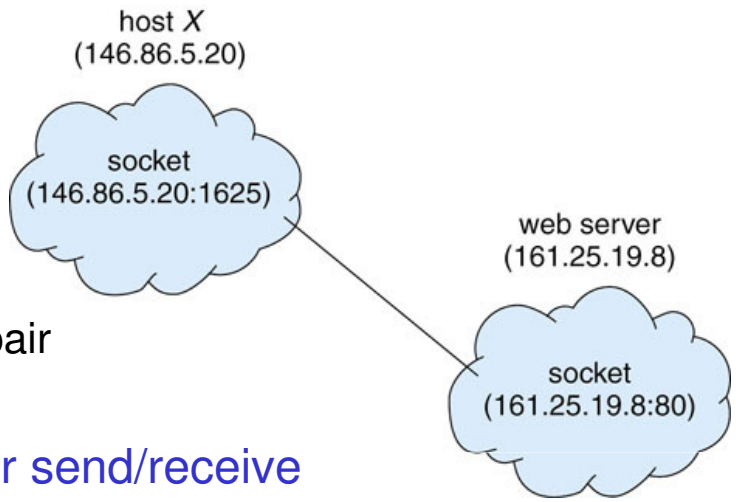
```
int fd[2];
pid_t child;

if (pipe (fd)<0) perror ("pipe");

child = fork();
if (child<0) perror ("fork");

if (child != 0) /* parent: writer */
      close (fd[0]);
      write (fd[1], ......);
else{ /* child: reader */
      close (fd[1]);
      read (fd[0], ......);
}
```

- The pair of file descriptors has a read (0) and a write (1) side that are connected.
  - closing one of the two on both sides creates a channel
- Synchronization:
  - *read()* blocks when pipe (buffer) is empty
  - *write()* blocks when buffer is full
  - can choose also non-blocking behavior



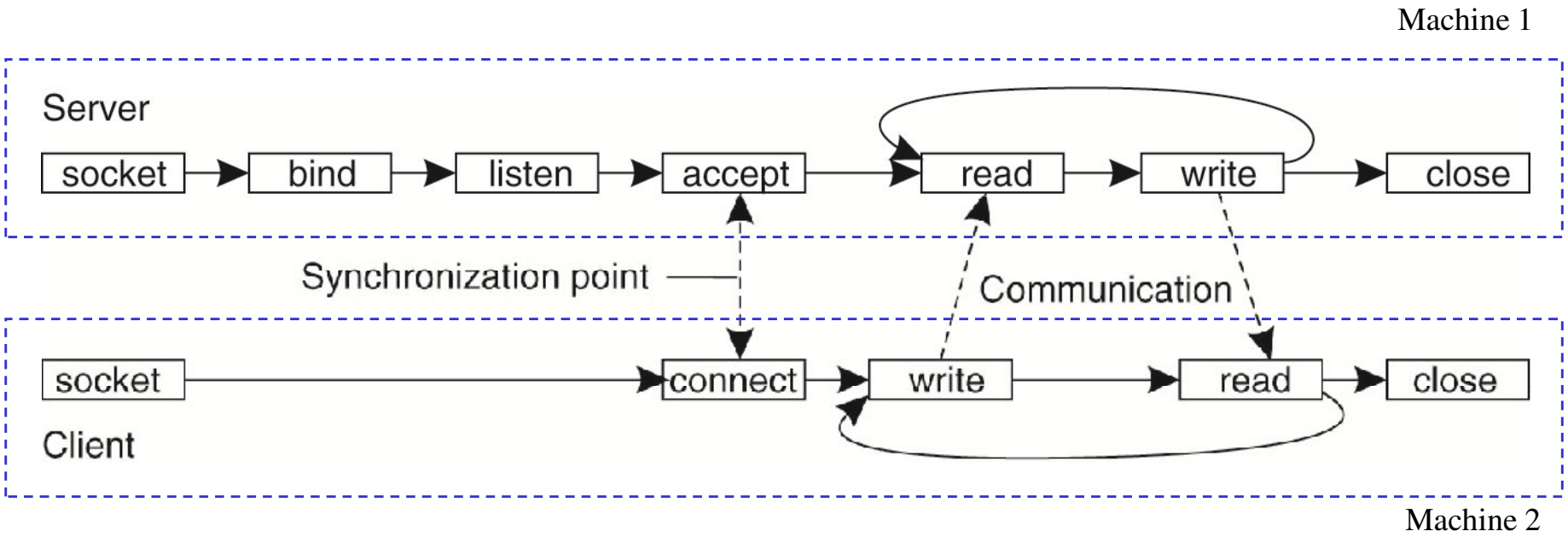
Distributed channels

- The Berkeley Socket Interface
  - setup a connection using *ports* on both machines
    - setup: client (active, trying) and server (passive, waiting) roles
    - endpoints identified by (machine, port) pair
    - connections identified by two such pairs
  - communicate using regular read/write or send/receive operations, similar as file operations
  - .... discussed in detail in 'computer networks'



Primitive	Meaning
SOCKET	Create a new communication end point
BIND	Attach a local address to a socket
LISTEN	Announce willingness to accept connections; give queue size
ACCEPT	Block the caller until a connection attempt arrives
CONNECT	Actively attempt to establish a connection
SEND	Send some data over the connection
RECEIVE	Receive some data from the connection
CLOSE	Release the connection

Usage of the interface (C binding)



- Note: send/receive is same as read/write

POSIX: message queues (1003.1b)

- A named priority queue of messages
  - *name*: system-wide
  - *priority*: support discrimination based on importance
  - *messages*: structure is maintained: a *receive()* is coupled with a *send()*

```
struct mq_attr attr;
mqd_t mq;

attr.mq_maxmsg = 100; attr.mq_msgsize = 128;
attr.mq_flags = 0;
mq = mq_open("/MyQ", how, mode, &attr);

if (mq == (mqd_t)-1) perror ("mq_open");

/* how: e.g. O_CREAT | O_RDWR
 * mode: e.g. S_IRUSR | S_IWUSR | S_IXUSR  */
```

POSIX: message queues (cnt'd)

- Manipulation through
  - *mq_setattr(), mq_getattr()*
    - queue size, message size – set upon creation only
    - current number of messages, (un)blocking use of communication functions
  - *mq_close(), mq_unlink()*
    - unlinking possible only after closing by all partners

- Communication:
  - *status = mq_send(mq, "hello world", 12, prio)*
    - *prio:* between 0 and MQ_PRIO_MAX (≥ 32)
  - *nbytes = mq_receive(mq, buf, maxsize, &prio)*
    - (oldest, highest prio) first

Message passing primitives properties

- Source / destination
  - send to / receive from process, or object
    - object: channel, mailbox
- Synchronization
  - blocking / non blocking
  - synchronous, buffered, asynchronous
    - respectively: send/receive simultaneously, excess of sends allowed, send never blocked
- Delivery order
  - fifo, arbitrary (receiver determined)
- Numbers of sender / receiver
  - point-to-point
  - multiple readers, writers
    - multicast: group of receivers
    - anycast: one receiver out of a group
- Structure
  - message oriented, byte oriented

Examples

- Pipe, socket
  - Object
  - Buffered, blocking
  - Fifo
  - Multiple writers
    - (to receiving side)
  - Byte oriented
- Mailbox
  - Object
  - Buffered, non-blocking
  - Receiver-determined order
  - Multiple readers, writers
  - Message oriented, usually