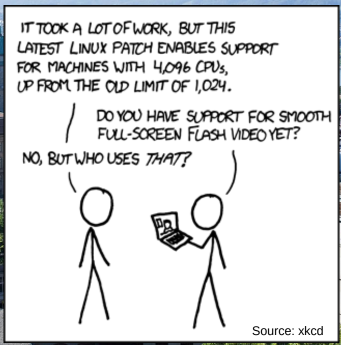
Adrian Perrig & Torsten Hoefler

Networks and Operating Systems (252-0062-00)
Chapter 4: Scheduling and Synchronization



Source: xkcd

Example: Linux o(1) scheduler

- **140 level Multilevel Feedback Queue**
  - 0-99 (high priority):
    static, fixed, "realtime"
    FCFS or RR
  - 100-139: User tasks, dynamic
    Round-robin within a priority level
    Priority ageing for interactive (I/O intensive) tasks
- **Complexity of scheduling is independent of no. tasks**
  - Two arrays of queues: "runnable" & "waiting"
  - When no more task in "runnable" array, swap arrays

Example: Linux "completely fair scheduler"

- **Task's priority = how little progress it has made**
  - Adjusted by fudge factors over time
  - Get "bonus" if a task yields early (his time is distributed evenly)
- **Implementation uses Red-Black tree**
  - Sorted list of tasks
  - Operations now O(log n), but this is fast
- **Essentially, this is the old idea of "fair queuing" from packet networks**
  - Also called "generalized processor scheduling"
  - Ensures guaranteed service rate for all processes
  - CFS does not, however, expose (or maintain) the guarantees

Problems with UNIX Scheduling

- **UNIX conflates protection domain and resource principal**
  - Priorities and scheduling decisions are per-process (thread)
- **However, may want to allocate resources across processes, or separate resource allocation within a process**
  - E.g., web server structure
    Multi-process
    Multi-threaded
    Event-driven
  - If I run more compiler jobs than you, I get more CPU time
- **In-kernel processing is accounted to nobody**

Resource Containers [Banga et al., 1999]

New OS abstraction for explicit resource management, separate from process structure

- **Operations to create/destroy, manage hierarchy, and associate threads or sockets with containers**
- **Independent of scheduling algorithms used**
- **All kernel operations and resource usage accounted to a resource container**

⇒ **Explicit and fine-grained control over resource usage**
⇒ **Protects against some forms of DoS attack**

- **Most obvious modern form: *virtual machines, containers***

Our Small Quiz

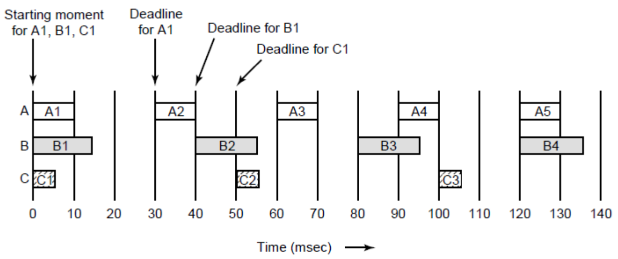
- **True or false (raise hand)**
  - Throughput is an important goal for batch schedulers
  - Response time is an important goal for batch schedulers
  - Realtime schedulers schedule jobs faster than batch schedulers
  - Realtime schedulers have higher throughput than batch schedulers
  - The scheduler has to be invoked by an application
  - SJF scheduling minimizes job waiting times
  - FCFS scheduling has low average waiting times
  - Starvation can occur in FCFS scheduling
  - Starvation can occur in SJF scheduling
  - Preemption can be used to improve interactivity
  - Round Robin scheduling is fair
  - Priority scheduling always suffers from starvation
  - $\Omega(\log N)$ is a lower time-bound for scheduling N processes
  - Multilevel Feedback Queues in Linux prevent starvation
  - Simple Unix scheduling fairly allocates the time to each user

Real Time

Real-time scheduling

- **Problem: giving real time-based guarantees to tasks**
  - Tasks can appear at any time
  - Tasks can have deadlines
  - Execution time is generally known
  - Tasks can be periodic or aperiodic
- **Must be possible to reject tasks which are unschedulable, or which would result in no feasible schedule**

Example: multimedia scheduling



Rate-monotonic scheduling

- **Schedule periodic tasks by always running task with shortest period first.**
  - Static (offline) scheduling algorithm
- **Suppose:**
  - m tasks
  - C_i is the execution time of i'th task
  - P_i is the period of i'th task
- **Then RMS will find a feasible schedule if:**

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq m(2^{1/m} - 1)$$

- **(Proof is beyond scope of this course)**

cf. Liu, Leiland: "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment", JACM 1973

Earliest deadline first

- **Schedule task with earliest deadline first (duh..)**
  - Dynamic, online.
  - Tasks don't *actually* have to be periodic…
  - More complex – at first sight O(n) for scheduling decisions
- **EDF will find a feasible schedule if:**

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

- **Which is very handy. Assuming zero context switch time…**

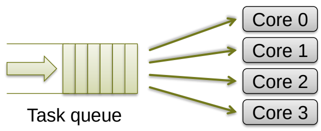
Guaranteeing processor rate

- **E.g., you can use EDF to guarantee a rate of progress for a long-running task**
  - Break task into periodic jobs, period *p* and time *s*.
  - A task arrives at start of a period
  - Deadline is the end of the period
- **Provides a *reservation* scheduler which:**
  - Ensures task gets *s* seconds of time every *p* seconds
  - Approximates weighted fair queuing
- **Algorithm is regularly rediscovered…**

Multiprocessor Scheduling

Challenge 1: sequential programs on multiprocessors

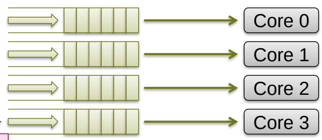
- **Queuing theory ⇒ straightforward, although:**
  - More complex than uniprocessor scheduling
  - Harder to analyze



but…

It's much harder

- **Overhead of locking and sharing queue**
  - Classic case of scaling bottleneck in OS design
- **Solution: per-processor scheduling queues**



In practice, each is more complex e.g., MFQ

It's much harder

- **Threads allocated arbitrarily to cores**
  - ⇒ tend to move between cores
  - ⇒ tend to move between caches
  - ⇒ really bad locality and hence performance
- **Solution: affinity scheduling**
  - Keep each thread on a core most of the time
  - Periodically rebalance across cores
  - Note: this is often *non-work-conserving!*
- **Alternative: hierarchical scheduling (Linux)**

Challenge 2: parallel applications

- **Global barriers in parallel applications ⇒ One slow thread has huge effect on performance**
  - Corollary of *Amdahl's Law*
- **Multiple threads would benefit from cache sharing**
- **Different applications pollute each others' caches**
- **Leads to concept of "co-scheduling"**
  - Try to schedule all threads of an application together
- **Critically dependent on *synchronization* concepts**

Multicore scheduling

- **Multiprocessor scheduling is two-dimensional**
  - When to schedule a task?
  - Where (which core) to schedule on?
- **General problem is NP hard ☹**
- **But it's worse than that:**
  - Don't want a process holding a lock to sleep
    ⇒ Might be other running tasks spinning on it
  - Not all cores are equal
- **In general, this is a wide-open research problem**

Little's Law

- **Assume, in a train station:**
  - 100 people arrive per minute
  - Each person spends 15 minutes in the station
  - How big does the station have to be (house how many people)
- **Little's law: "*The average number of active tasks in a system is equal to the average arrival rate multiplied by the average time a task spends in a system*"**

Last time: Scheduling

- **Basics:**
  - Workloads, tradeoffs, definitions
- **Batch-oriented scheduling**
  - FCFS, Convoys, SJF, Preemption: SRTF
- **Interactive workloads**
  - RR, Priority, Multilevel Feedback Queues, Linux, Resource containers
- **Realtime**
  - RMS, EDF
- **Multiprocessors**

- **This time: OSPP Section 5 (not including IPC)**

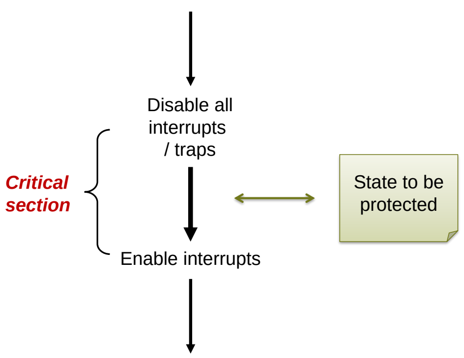
Goals today

- **Overview of inter-process communication systems**
  - Hardware support
  - With shared memory
  - Without shared memory
  - Upcalls

- **Generally: very broad field**
  - Quite competitive… especially with microkernels

Recap: Hardware support for synchronization

Disabling interrupts



Disabling interrupts

- **Nice and simple**
- **Can't be rescheduled inside critical section**
  ⇒ data can't be altered by anything else
- **Except…**
- **Another processor!**
  - Hmm….

- **Very efficient if in kernel on a *uniprocessor*.**

Test-And-Set instruction

- **Atomically:**
  - Read the value of a memory location
  - Set the location to 1 (or another constant)
- **Available on some hardware (e.g., PA-RISC)**
  - (actually, more a RAC – Read-And-Clear)

Compare-And-Swap (CAS)

```
word cas(word *flag, word oldval, word newval) {
        atomically {
                if (*flag == oldval) {
                        *flag = newval;
                        return oldval;
                } else {
                        return *flag;
                }
        }
}
```

- **Available on e.g., x86, IBM/370, SPARC, ARM,…**
- **Theoretically, *slightly* more powerful than TAS**
  - Why?
  - Other variants e.g., CAS2, etc.

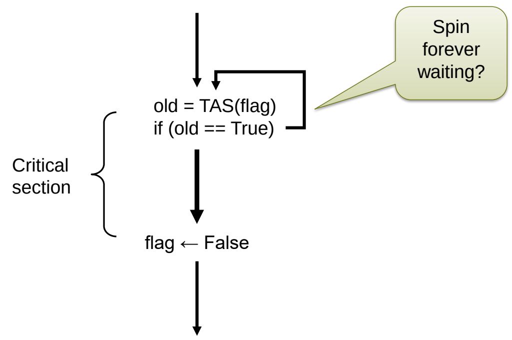
Load-Link, Store-Conditional

Factors `CAS`, etc. into two instructions:

1. **`LL`: load from a location and mark as "owned"**
2. **`SC`: Atomically:**
   1. Store *only* if already marked by this processor
   2. Clear any marks set by other processors
   3. Return whether it worked.

Available on PPC, Alpha, MIPS, etc…

Back to TAS…



Spinning

- **On a uniprocessor:**
  - Not much point in spinning at all. What's going to happen?
  - Possibly an interrupt
- **On a multiprocessor:**
  - Can't spin forever
  - Another spin is always cheap
  - Blocking thread and rescheduling is expensive
  - Spinning only works if lock holder is running on another core

Competitive spinning

- **How long to spin for?**
- **"Competitive spinning":**
  - Within a factor of 2 of optimal, offline (i.e., impossible!) algorithm
- **Good approach: spin for the context switch time**
  - Best case: avoid context switch entirely
  - Worst case: twice as bad as simply rescheduling

IPC with shared memory

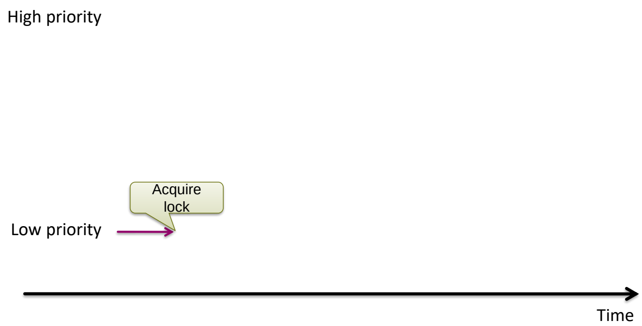
Techniques you already know ☺

- **Semaphores**
  - P, V operations
- **Mutexes**
  - Acquire, Release
- **Condition Variables**
  - Wait, Signal (Notify), Broadcast (NotifyAll)
- **Monitors**
  - Enter, Exit

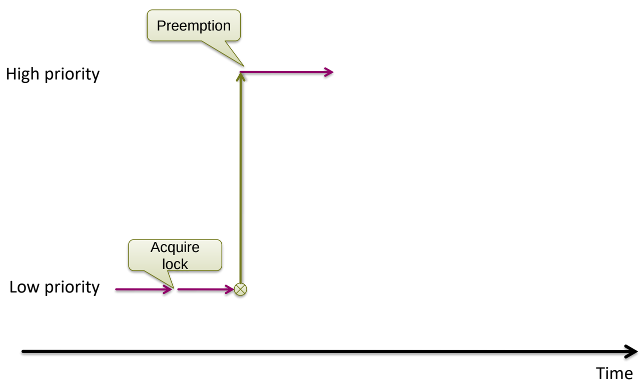
Focus here: interaction with scheduling

- **Most OSes provide some form of these**
- **Key issue not yet covered: interaction between scheduling and synchronization**
- **Example: Priority inversion**
  - Assuming a priority scheduler, e.g., Unix, Windows

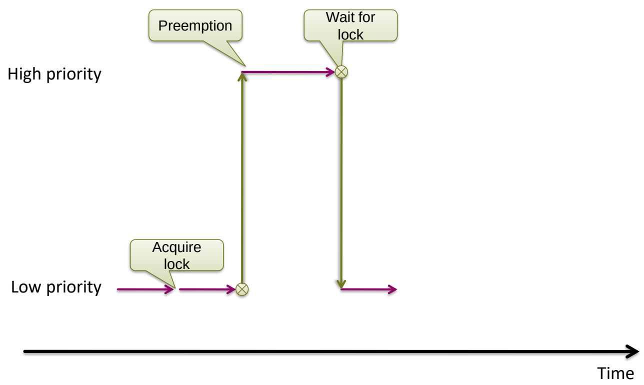
Priority Inversion



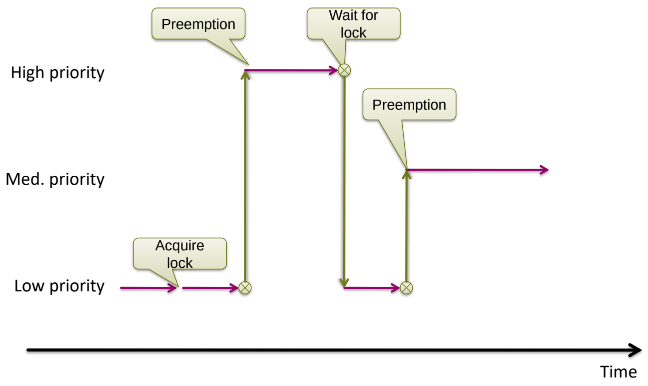
Priority Inversion



Priority Inversion

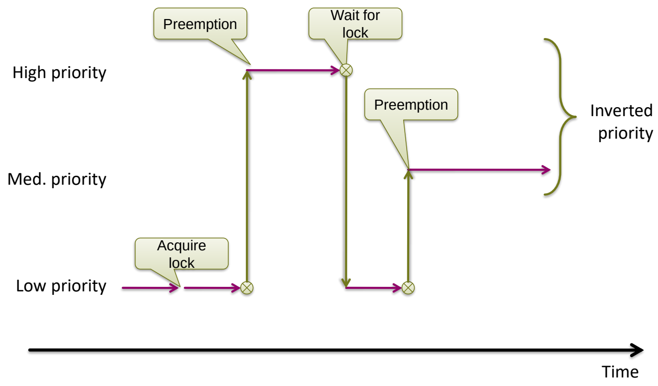


Priority Inversion



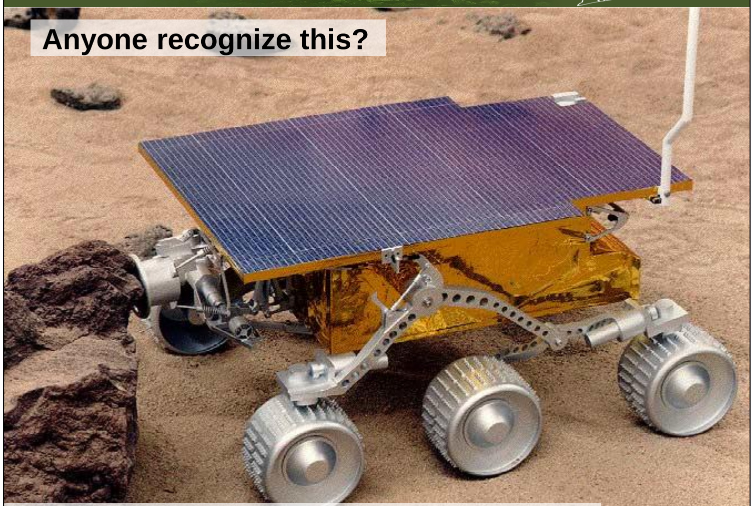
ETH zürich spcl.inf.ethz.ch @spcl_eth

Priority Inversion



ETH zürich spcl.inf.ethz.ch @spcl_eth

Anyone recognize this?



http://research.microsoft.com/en-us/um/people/mbj/Mars_Pathfinder/Mars_Pathfinder.html

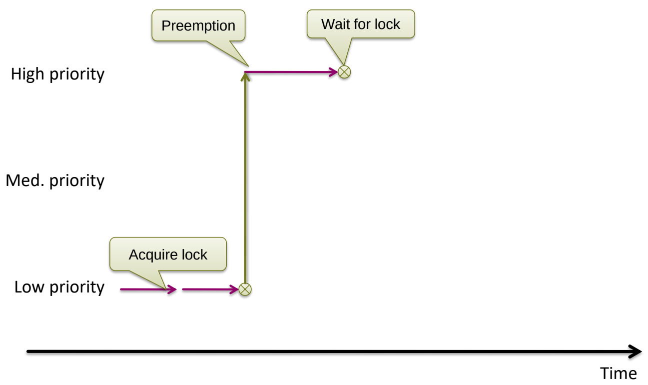
ETH zürich spcl.inf.ethz.ch @spcl_eth

Priority Inheritance

- **Process holding lock *inherits* priority of highest priority process that is waiting for the lock.**
  - Releasing lock ⇒ priority returns to previous value
  - Ensures forward progress
- **Alternative: *Priority Ceiling***
  - Process holding lock acquires priority of highest-priority process that *can ever* hold lock
  - Requires static analysis, used in embedded RT systems

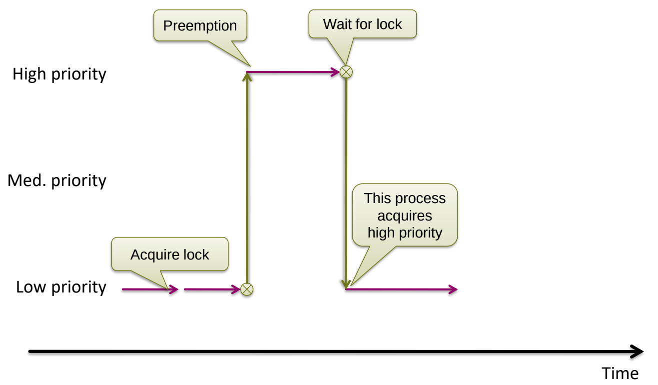
ETH zürich spcl.inf.ethz.ch @spcl_eth

Priority Inheritance

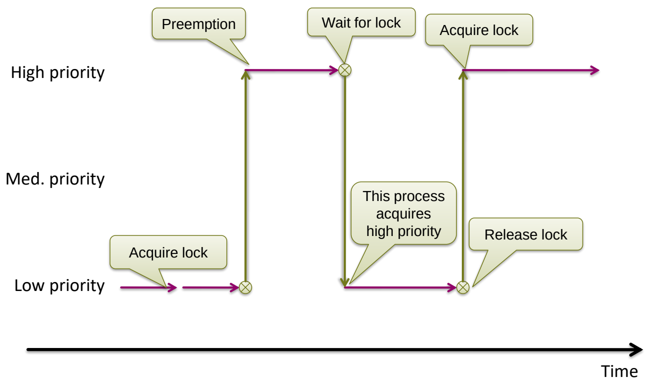


ETH zürich spcl.inf.ethz.ch @spcl_eth

Priority Inheritance

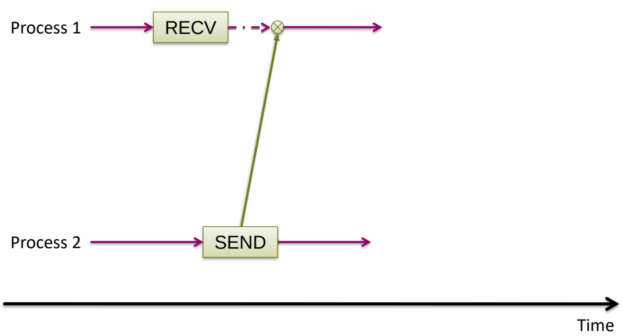


Priority Inheritance

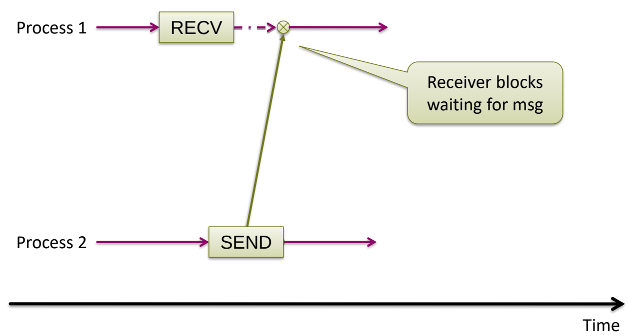


IPC without shared memory

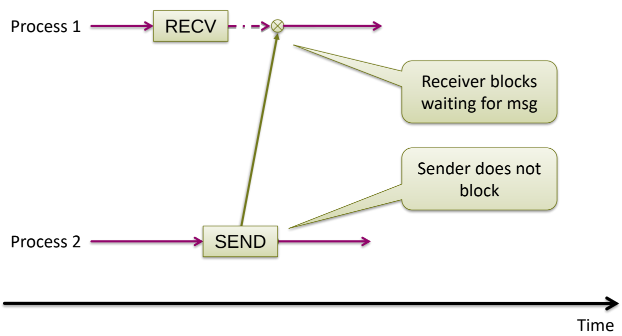
Asynchronous (buffered) IPC



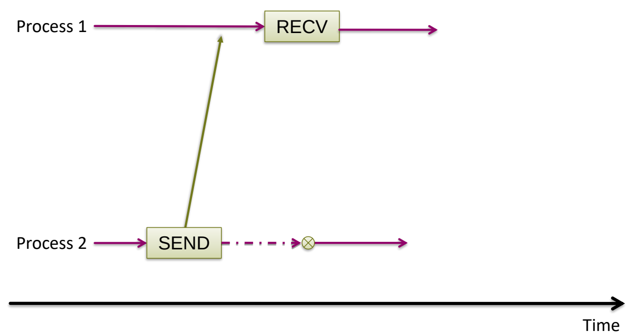
Asynchronous (buffered) IPC



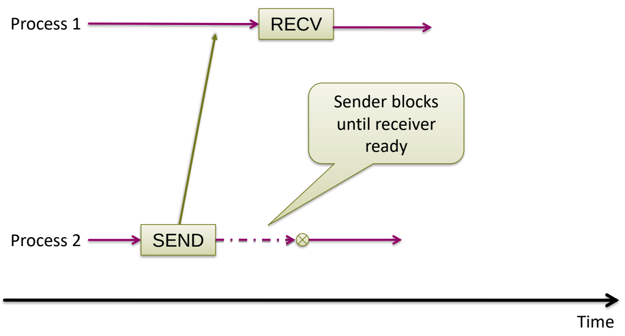
Asynchronous (buffered) IPC



Synchronous (unbuffered) IPC



Synchronous (unbuffered) IPC



Duality of messages and shared-memory

- **Famous claim by Lauer and Needham (1978):**

Any shared-memory system (e.g., one based on monitors and condition variables) is equivalent to a non-shared-memory system (based on messages)

- **Exercise: pick your favourite example of one, and show how to build the dual.**

Unix Pipes

- **Basic (first) Unix IPC mechanism**
- **Unidirectional, buffered communication channel between two processes**
- **Creation:**

  `int pipe(int pipefd[2])`

- **Q. How to set up pipe between two processes?**
- **A. Don't! Create the pipe first, then fork…**

Pipe idiom (man 2 pipe)

```
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Create a pipe

Pipe idiom (man 2 pipe)

```
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Fork

Pipe idiom (man 2 pipe)

```
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

In child: close write end

Pipe idiom (man 2 pipe)

```
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

Read from pipe and write to standard output until EOF

Pipe idiom (man 2 pipe)

```
int
main(int argc, char *argv[])
{
    int pipefd[2];
    pid_t cpid;
    char buf;

    assert(argc == 2);

    if (pipe(pipefd) == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }

    cpid = fork();
    if (cpid == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
    }

    if (cpid == 0) {    /* Child reads from pipe */
        close(pipefd[1]);          /* Close unused write end */

        while (read(pipefd[0], &buf, 1) > 0)
            write(STDOUT_FILENO, &buf, 1);

        write(STDOUT_FILENO, "\n", 1);
        close(pipefd[0]);
        _exit(EXIT_SUCCESS);

    } else {            /* Parent writes argv[1] to pipe */
        close(pipefd[0]);          /* Close unused read end */
        write(pipefd[1], argv[1], strlen(argv[1]));
        close(pipefd[1]);          /* Reader will see EOF */
        wait(NULL);                /* Wait for child */
        exit(EXIT_SUCCESS);
    }
}
```

In parent: close read end and write argv[1] to pipe

Unix shell pipes

- **E.g.:**

```
curl --silent http://spcl.inf.ethz.ch/Teaching/2015-osnet/ | sed
   's/[^A-Za-z]/\n/g' | sort -fu | egrep -v '^\s*$' | wc -l
```

- **Shell forks each element of the pipeline**
  - Each process connected via pipes
  - Stdout of process *n* → stdin of process *n+1*
  - Each process then exec's the appropriate command
  - Exercise: write it! (hint: 'man dup2'…)

Messaging systems

- **A good textbook will examine options:**
  - End-points may or may not know each others' names
  - Messages might need to be sent to more than one destination
  - Multiple arriving messages might need to be demultiplexed
  - Can't wait forever for one particular message
- **BUT: you'll see most of this somewhere else!**
  - In networking
  - Many parallels between message-passing operating systems and networks

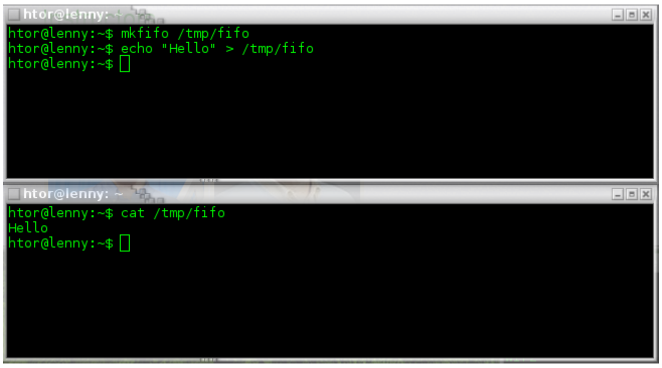
Example

- **The concept of a "port" allows:**
  - Naming of different end-points within a process
  - Demultiplexing of messages
  - Waiting selectively for different kinds of messages
- **Analogous to "socket" and "TCP port" in IPv4**
  - In Unix, "Unix domain sockets" do exactly this.
  - `int s = socket(AF_UNIX, type, 0);`

Naming pipes

- **Pipes so far are only named by their descriptors**
  - Namespace is *local* to the process
  - Copied on `fork()`
- **How to put a pipe in the global namespace?**
  - Make it a "named pipe"
  - Special file of type "pipe" (also known as a FIFO)

Named pipes



Local Remote Procedure Call

- **Can use RPC locally:**
  - Define procedural interface in an IDL
  - Compile / link stubs
  - Transparent procedure calls over messages
- **Naïve implementation is slow**
  - Lots of things (like copying) don't matter with a network, but do matter between local processes
  - Can be made very fast: more in the AOS course…

Unix signals

- ***Asynchronous* notification from the kernel**
- **Receiver doesn't wait: signal just happens**
- **Interrupt process, and:**
  - Kill it
  - Stop (freeze) it
  - Do "something else" (see later)

Signal types (some of them)

Name	Description / meaning	Default action
SIGHUP	Hangup / death of controlling process	Terminate process
SIGINT	Interrupt character typed (CTRL-C)	e process
SIGQUIT	Quit character typed (CTRL-\)	Core dump
SIGKILL	`kill -9 <process id>`	**Terminate process**
SIGSEGV	Segfault (invalid memory referenc	ump
SIGPIPE	Write on pipe with no reader	Terminate process
SIGALRM	`alarm()` goes off	inate process
SIGCHLD	Child process stopped or terminated	Ignored
SIGSTOP	Stop process	Stop
SIGCONT	Continue process	
SIGUSR1,2	User-defined signals	Terminate process

"Hanging up" the phone (terminal)

Can't be disabled!

E.g., after other side of pipe has closed it

Used by debuggers (e.g., `gdb`) and shell (`CTRL-Z`)

Etc. – see `man 7 signal` for the full list

Where do signals come from?

- **Memory management subsystem:**
  - `SIGSEGV`, etc.
- **IPC system**
  - `SIGPIPE`
- **Other user processes**
  - `SIGUSR1,2`, `SIGKILL`, `SIGSTOP`, `SIGCONT`
- **Kernel trap handlers**
  - `SIGFPE`
- **The "TTY Subsystem"**
  - `SIGINT`, `SIGQUIT`, `SIGHUP`

Sending a signal to a process

- **From the Unix shell:**

```
$ kill –HUP 4234
```

- **From C:**

```
#include <signal.h>
int kill(pid_t pid, int signo);
```

- **"Kill" is a rather unfortunate name ☹**

Unix signal handlers

- **Change what happens when a signal is delivered:**
  - Default action
  - Ignore signal
  - Call a user-defined function in the process
    → the ***signal handler***
- **Allows signals to be used like "user-space traps"**

Oldskool: `signal()`

- **Test your C parsing skills:**

```
#include <signal.h>

void (*signal(int sig, void (*handler)(int))) (int);
```

- **What does this mean?**

Oldskool: `signal()`

```
void (*signal(int sig, void (*handler)(int))) (int);
```

- **Unpacking this:**
  - A handler looks like
    `void my_handler(int);`
  - Signal takes two arguments…
    An integer (the signal type, e.g. `SIGPIPE`)
    A pointer to a handler function
  - … and returns a pointer to a handler function
    The previous handler,
- **"Special" `handler` arguments:**
  - `SIG_IGN` (ignore), `SIG_DFL` (default), `SIG_ERR` (error code)

Unix signal handlers

- **Signal handler can be called at *any time!***
- **Executes on the current user stack**
  - If process is in kernel, may need to retry current system call
  - Can also be set to run on a different (alternate) stack

⇒ **User process is in *undefined* state when signal delivered**

Implications

- **There is very little you can safely do in a signal handler!**
  - Can't safely access program global or static variables
  - Some system calls are *re-entrant*, and can be called
    including signal() and sigaction()
    full list see in "man 7 signal"
  - Many C library calls cannot (including `_r` variants!)
  - Can sometimes execute a `longjmp` if you are careful
- **What happens if another signal arrives?**

Multiple signals

- **If multiple signals of the *same* type are to be delivered, Unix will *discard all but one.***
- **If signals of *different* types are to be delivered, Unix will deliver them *in any order.***
- **Serious concurrency problem:**
  How to make sense of this?

A better `signal()` POSIX `sigaction()`

```
#include <signal.h>

int sigaction(int signo,
              const struct sigaction *act,
              struct sigaction *oldact);

struct sigaction {
      void (*sa_handler)(int);
      sigset_t    sa_mask;
      int         sa_flags;
      void (*sa_sigaction)(int, siginfo_t *, void *);
};
```

New action for signal `signo`

Previous action is returned

Signal handler

Signals to be blocked in this handler (cf., `fd_set`)

More sophisticated signal handler (depending on flags)

Signals as upcalls

- **Particularly specialized (and complex) form of an *upcall***
  - Kernel RPC to user process
- **Other OSes use upcalls much more heavily**
  - Including Barrelfish
  - "Scheduler Activations": dispatch every process using an upcall instead of return
- ***Very* important structuring concept for systems!**