Real-Time Programming & RTOS

Concurrent and real-time programming tools

Concurrent Programming

Concurrency in real-time systems

- typical architecture of embedded real-time system:
  - several input units
  - computation
  - output units
  - data logging/storing
- i.e., handling several concurrent activities
- concurrency occurs naturally in real-time systems

Support for concurrency in programming languages (Java, Ada, ...)
advantages: readability, OS independence, checking of interactions by compiler, embedded computer may not have an OS

Support by libraries and the operating system (C/C++ with POSIX)
advantages: multi-language composition, language's model of concurrency may be difficult to implement on top of OS, OS API stadards imply portability

Processes and Threads

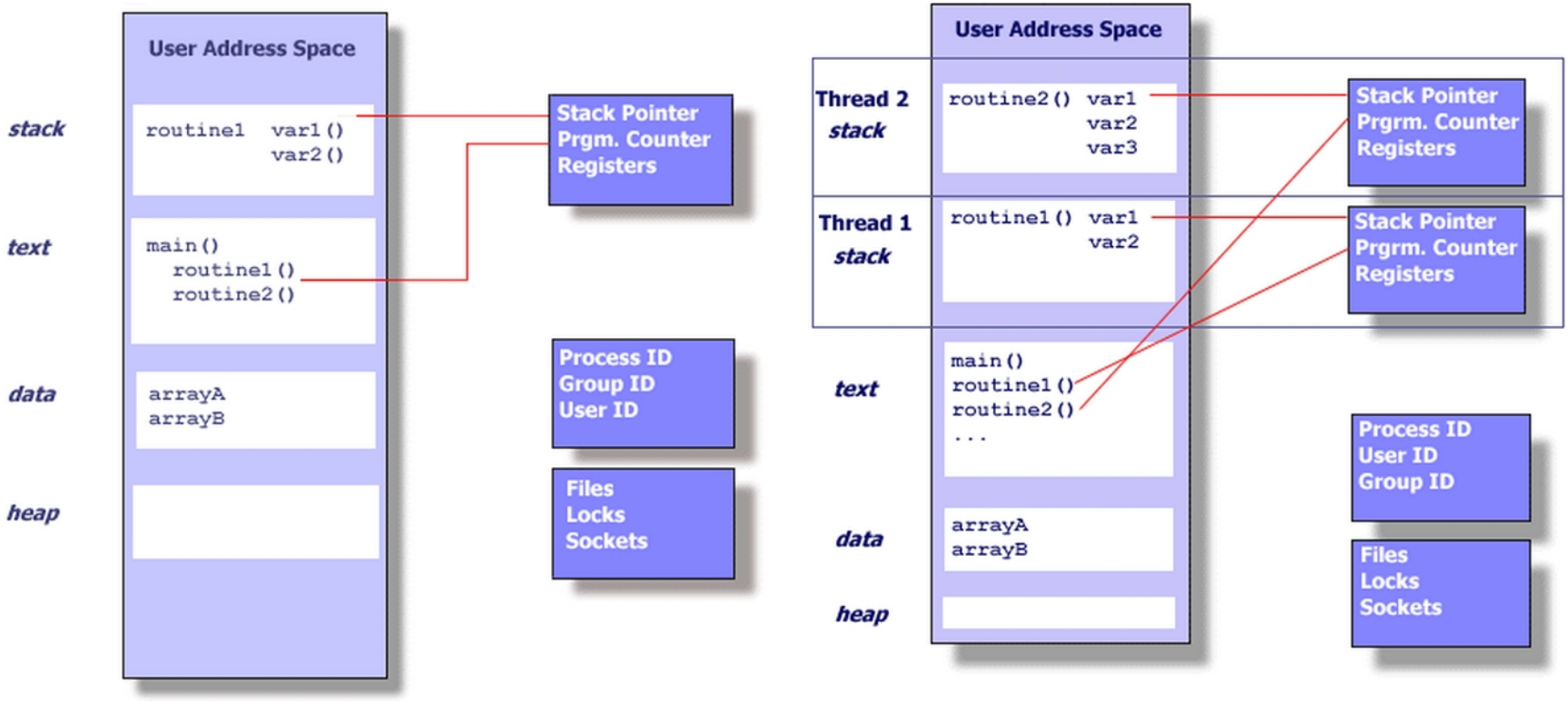
Process

- running instance of a program,
- executes its own virtual machine to avoid interference from other processes,
- contains information about program resources and execution state, e.g.:
  - environment, working directory, ...
  - program instructions,
  - registers, heap, stack,
  - file descriptors,
  - signal actions, inter-process communication tools (pipes, message boxes, etc.)

Thread

- exists *within a process*, uses process resources ,
- can be scheduled by OS and run as an independent entity,
- keeps its own: execution stack, local data, etc.
- share global data and resources with other threads of the same process

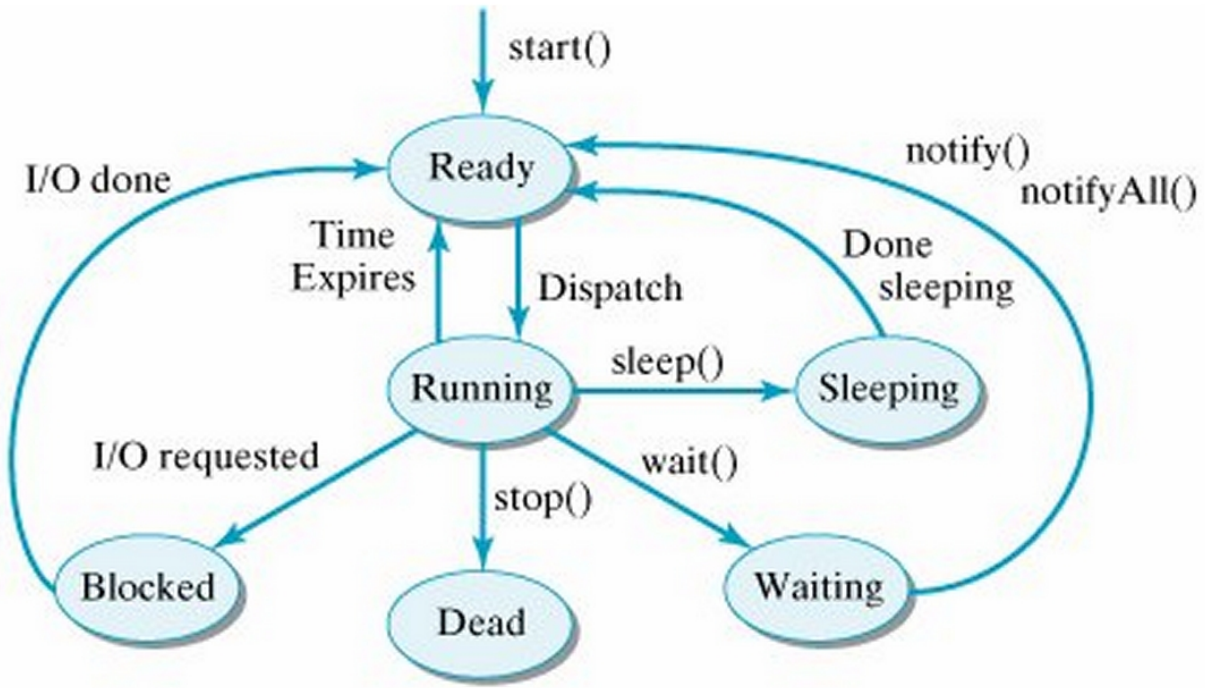
Processes and threads in UNIX



Threads: Resource Sharing

- changes made by one thread to shared system resources will be seen by all other threads
- two pointers having the same value point to the same data
- reading and writing to the same memory locations is possible, and therefore requires explicit synchronization by the programmer

Process (Thread) States



Process (Thread) Initialization and Termination

Initialization

- explicit process (thread) declaration
- fork and join
- cobegin, coend

Termination

- *completion* of execution
- “suicide” by execution of a *self-terminte* statement
- abortion, through the explicit action of *another process (thread)*
- ocurrence of an *error* condition
- never (process is a *non-terminating* loop)

Interprocess relations

- *parent-child*: a parent is a process (thread) that created a child: child inherits most attributes from its parent; parent may wait for a state change of a child

Concurrent Programming is Complicated

Multi-threaded applications with shared data may have numerous flaws

- **Race condition**

  Two or more threads try to access the same shared data, the result depends on the exact order in which their instructions are executed
- **Deadlock**

  occurs when two or more threads wait on each other, forming a cycle and preventing all of them from making any forward progress
- **Starvation**

  an idefinite delay or permanent blocking of one or more runnable threads in a multithreaded application
- **Livelock**

  occurs when threads are scheduled but are not making forward progress because they are continuously reacting to each other's state changes

Usually difficult to find bugs and verify correctness

Communication and Synchronization

Communication

- passing of information from one process to another
- typical methods: shared variables, message passing

Synchronization

- satisfaction of constraints on the interleaving of actions of processes

  e.g. action of one process has to occur after an action of another one
- typical methods: semaphores, monitors

Communication and synchronization are linked:

- communication requires synchronization
- synchronization corresponds to communication without content

Communication: Shared Variables

Consistency problems:

- unrestricted use of shared variables is unreliable
- *multiple update problem*
  example: shared variable X, assignment $X := X + 1$
  - load the current value of X into a register
  - increment the value of the register
  - store the value of the register back to X
- two processes executing these instruction $\Rightarrow$ certain interleavings can produce inconsistent results

Solution:

- parts of the process that access shared variables must be executed *indivisibly* with respect to each other
- these parts are called *critical section*
- required protection is called *mutual exclusion*

... one may use a special mutual ex. protocol (e.g. Peterson) or a synchronization mechanism – semaphores, monitors

Synchronization: Semaphores

A sempahore contains an integer variable that, apart from initialization, is accessed only through two standard operations: `wait()` and `signal()`.

- semaphore is initialized to a non-negative value (typically 1)
- `wait()` operation: decrements the semaphore value if the value is positive; otherwise, if the value is zero, the caller becomes blocked
- `signal()` operation: increments the semaphore value; if the value is not positive, then one process blocked by the semaphore is unblocked (usually in FIFO order)
- both `wait` and `signal` are atomic

Semaphores are elegant low-level primitive but error prone and hard to debug (deadlock, missing signal, etc.)

Synchronization: Monitors

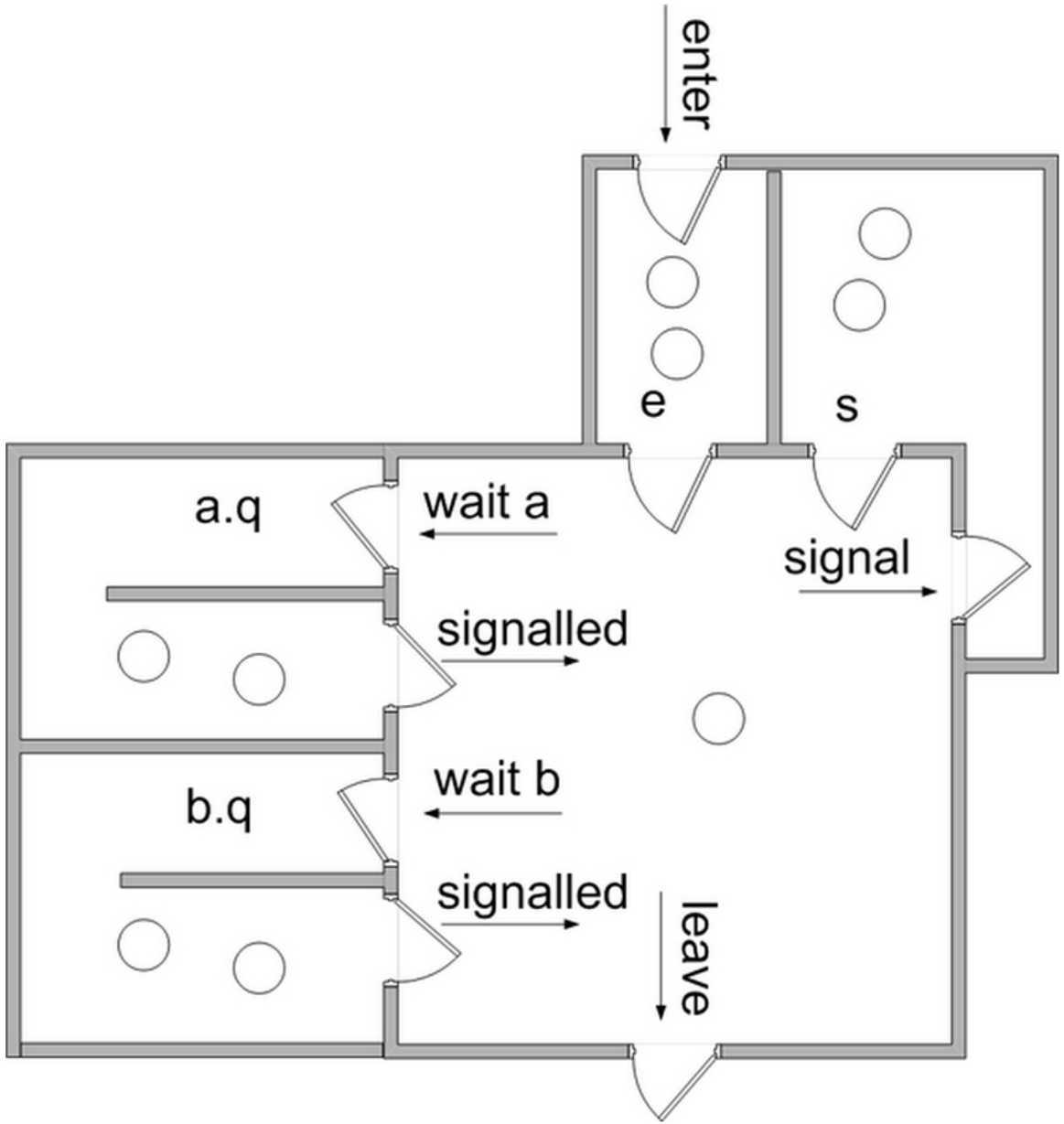
- *encapsulation* and efficient condition synchronization
- critical regions are written as procedures; all encapsulated in a single object or module
- procedure calls into the module are guaranteed to be mutually exclusive
- shared resources accessible only by these procedures

In some cases processes may need to wait until some condition holds true. The condition may be made true by another process using the monitor.

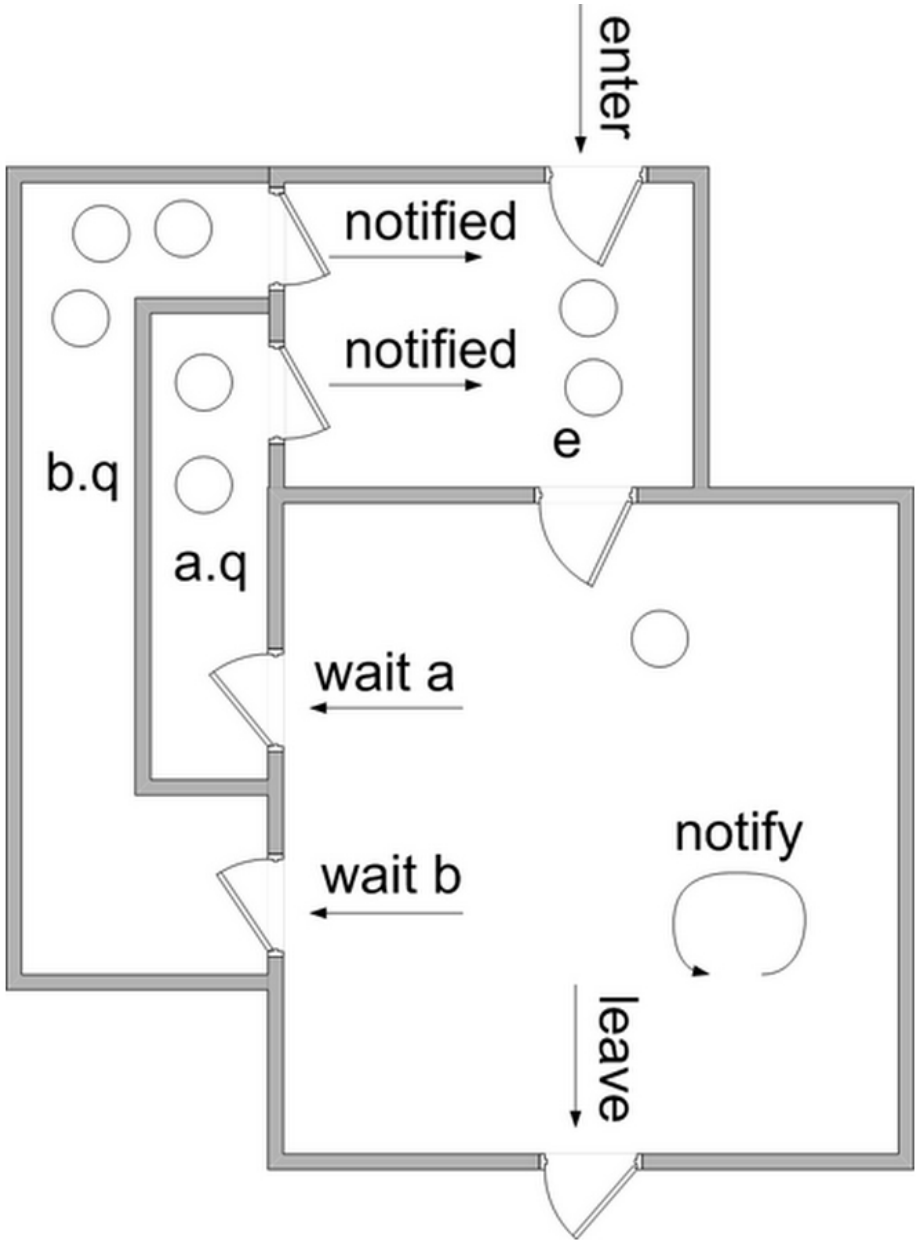
Solution: *condition variables*

- only two operations can be invoked on a condition variable `x` :
  - `x.wait()` = calling process is suspended until another process invokes `x.notify()`
  - `x.notify()` = resumes exactly one waiting process
- What happens when a process *P* notifies a process *Q* ?

Synchronization: Monitors – Hoare Style



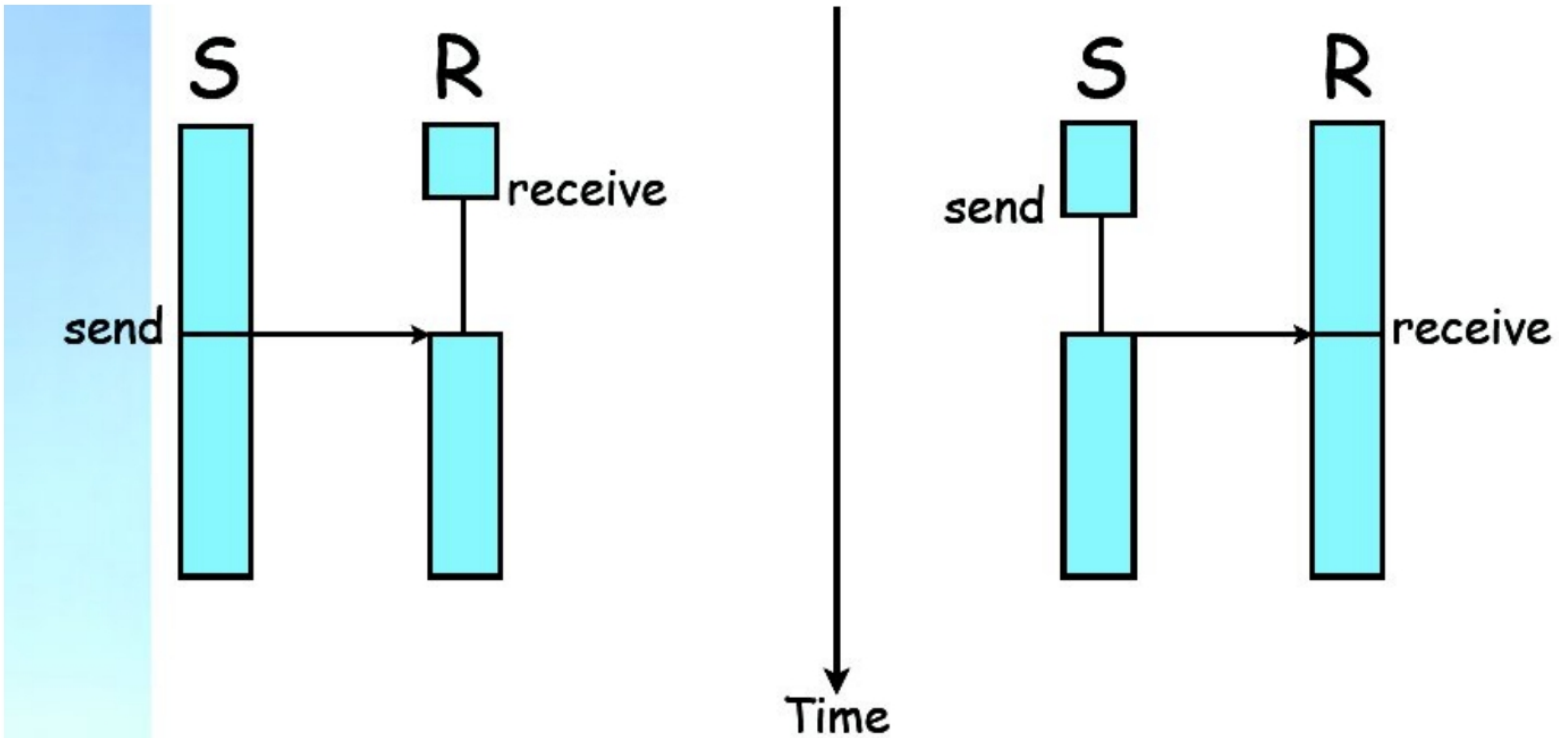
Synchronization: Monitors – Mesa Style



Communication: Message Passing

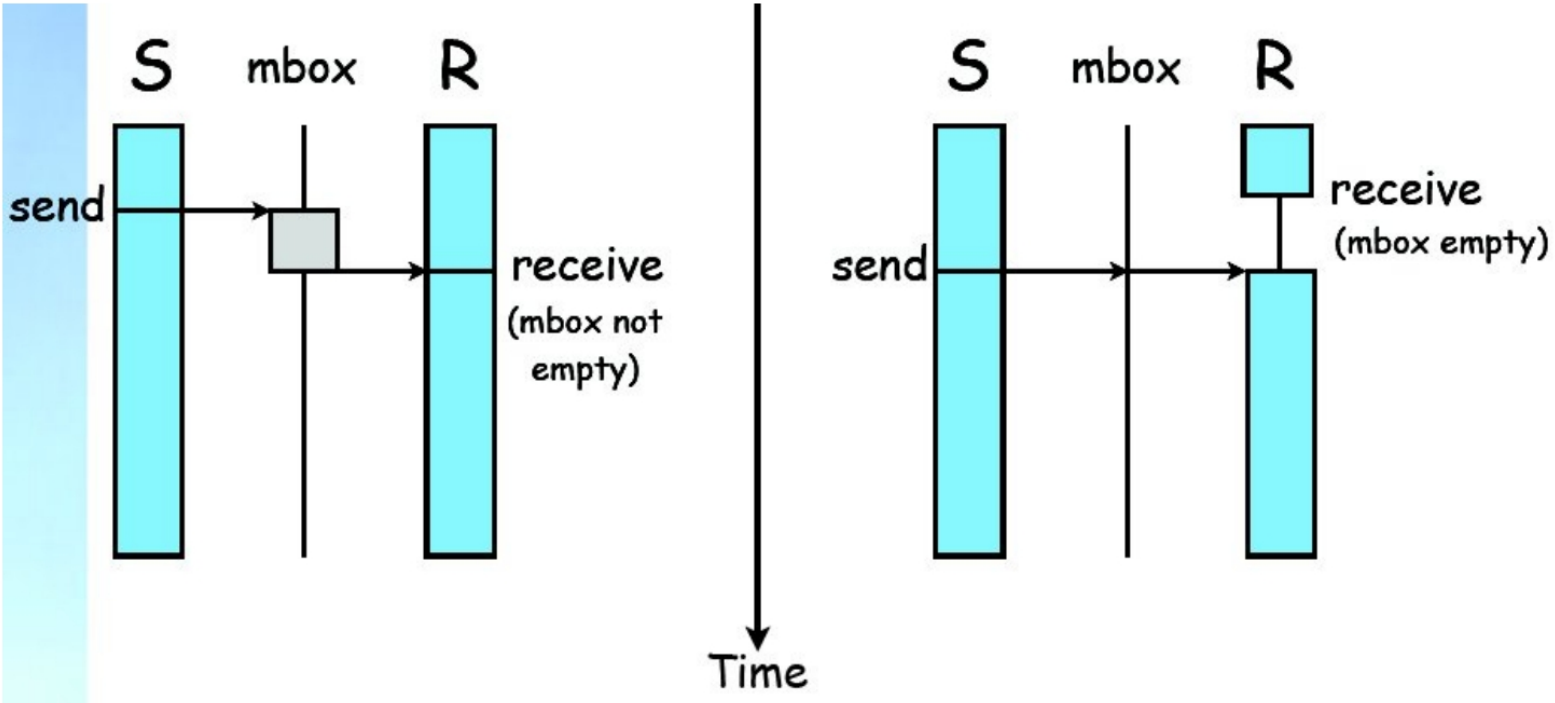
- **asynchronous** (no-wait): send operation is not blocking, requires buffer space
- **synchronous** (rendezvous): send operation is blocking, no buffer required
- **remote invocation** (extended rendezvous): sender is blocked until reply is received

Synchronous Message Passing



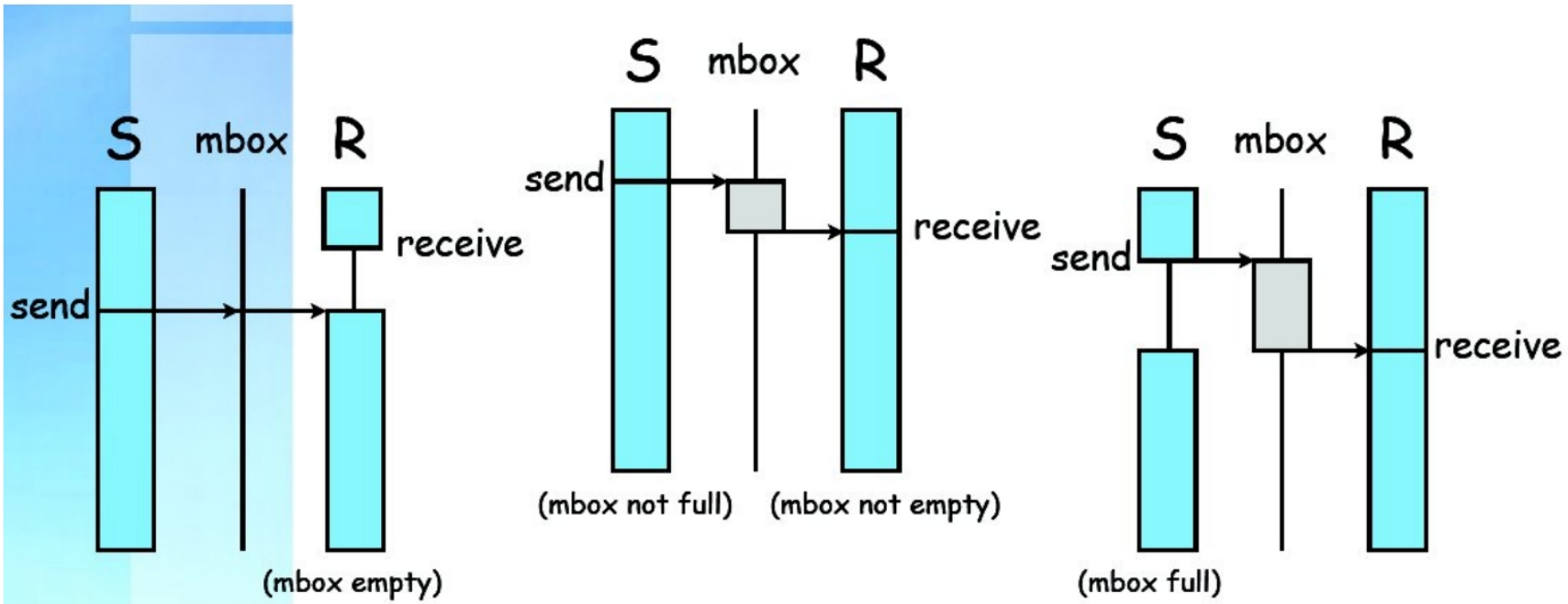
Both send and receive may block indefinitely!

Asynchronous Message Passing



Only the receiver might block indefinitely!

Asynch. Message Passing with Bounded Buffer



Receiver blocks if mbox is empty,
sender blocks if mbox is full

Note: size = 0 gives synchronous communication!

Process Naming

(In)direction

- *Direct*: process wanting to communicate must explicitly name the recipient or sender of the communication

  `send <message> to <process-name>`

- *Indirect*: messages sent to and received from mailboxes

  `send <message> to <mailbox>`

Direct naming has the advantage of simplicity, whilst indirect naming aids the decomposition of the software

Symmetry

- Symmetric: both sender and receiver name each other (directly or indirectly)

  `send <msg> to <process-name>`, `send <msg> to <mailbox>`

  `wait <msg> from <process-name>`, `wait <msg> from <mailbox>`

- Asymmetric: receiver names no specific source

  `wait <msg>`

Real-Time Aspects

- **time-aware systems** make explicit references to the time frame of the enclosing environment

  e.g. a bank safe's door are to be locked from midnight to nine o'clock

  - the "real-time" of the environment must be available

- **reactive systems** are typically concerned with relative times

  an output has to be produced within 50 ms of an associated input

  - must be able to measure intervals
  - usually must synchronize with environment: input sampling and output signalling must be done very regularly with controlled variability

The Concept of Time

Real-time systems must have a concept of time – but what is time?

- Measure of a time interval
  - Units?

    seconds, milliseconds, cpu cycles, system "ticks"
  - Granularity, accuracy, stability of the clock source
    - Is "one second" a well defined measure?
      "A second is the duration of 9,192,631,770 periods of the radiation corresponding to the transition between the two hyperfine levels of the ground state of the caesium-133 atom."
    - ... temperature dependencies and relativistic effects (the above definition refers to a caesium atom at rest, at mean sea level and at a temperature of 0 K)
  - Skew and divergence between multiple clocks

    Distributed systems and clock synchronization
- Measuring time
  - external source (GPS, NTP, etc.)
  - internal – hardware clocks that count the number of oscillations that occur in a quartz crystal

Requirements for Interaction with "time"

For RT programming, it is desirable to have:

- access to clocks and representation of time
- delays
- timeouts
- deadline specification and real-time scheduling

Access to Clock and Representation of Time

- requires a hardware clock that can be read like a regular external device
- mostly offered by an OS service, if direct interfacing to the hardware is not allowed

Example of time representation

(POSIX high resolution clock, counting seconds and nanoseconds since 1970 with known resolution)

```
#include <sys/time.h>

struct timespec {
        time_t   tv_sec;
        long     tv_nsec;
};
int clock_gettime(CLOCK_REALTIME, struct timespec *t);
int clock_getres (CLOCK_REALTIME, struct timespec *r);
```

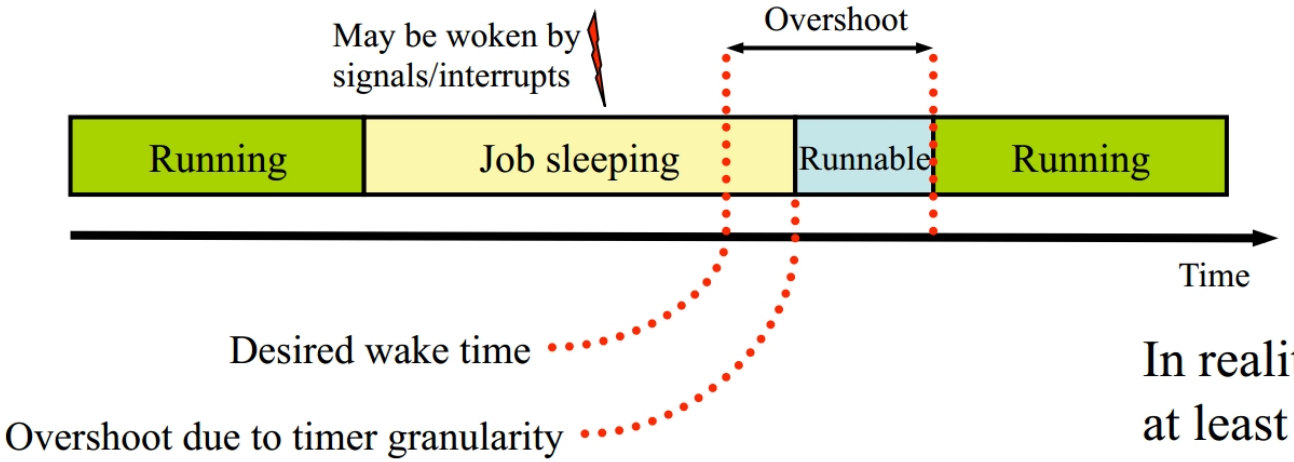
Delays

In addition to having access to a clock, need ability to

- Delay execution until an arbitrary calendar time

  What about daylight saving time changes? Problems with leap seconds.

- Delay execution for a relative period of time
  - Delay for *t* seconds



In reality: delay at least *t* seconds

  - Delay for *t* seconds after event *e* begins

```
start = curr_time();
do_action1();
delay(10.0 - (curr_time() - start));
do_action2();
```

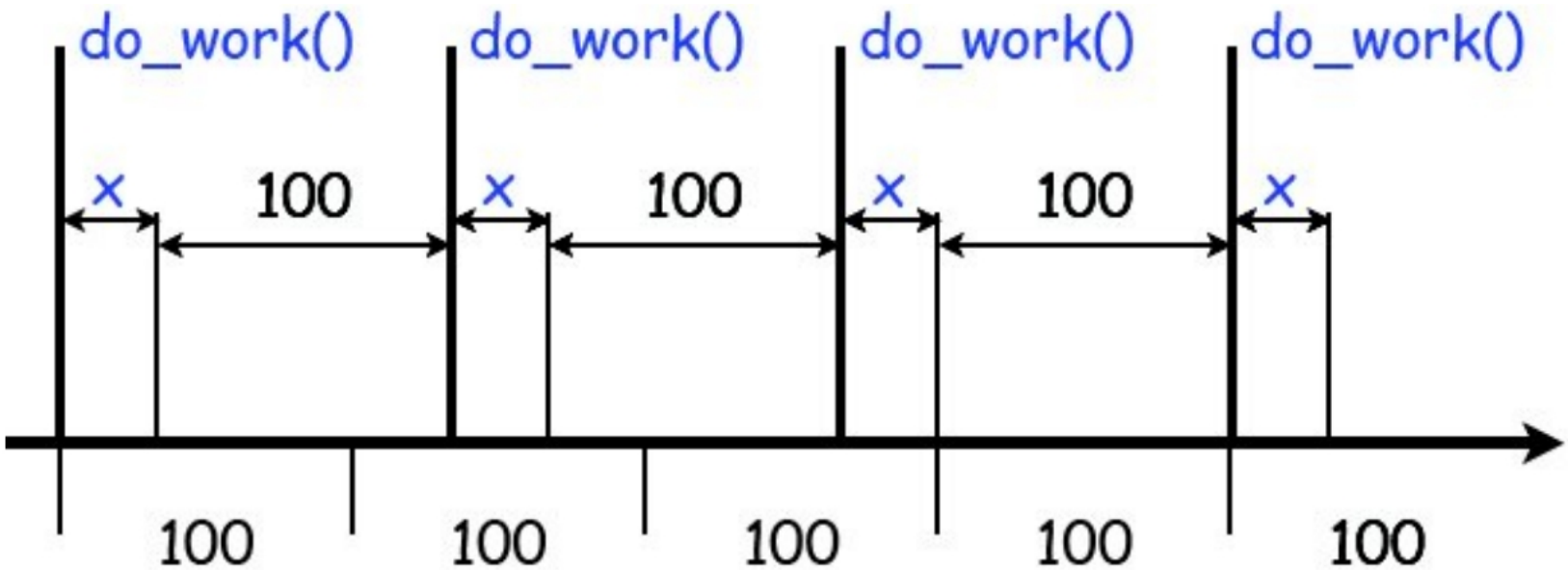
What if pre-empted between these?
Oversleep unless system has a function
`delay_until(start+10.0)`

A Repeated Task (An Attempt)

The goal is to `do work` repeatedly every 100 time units

```
while(1) {
  delay(100);
  do_work();
}
```

Does it work as intended? No, accumulates drift ...



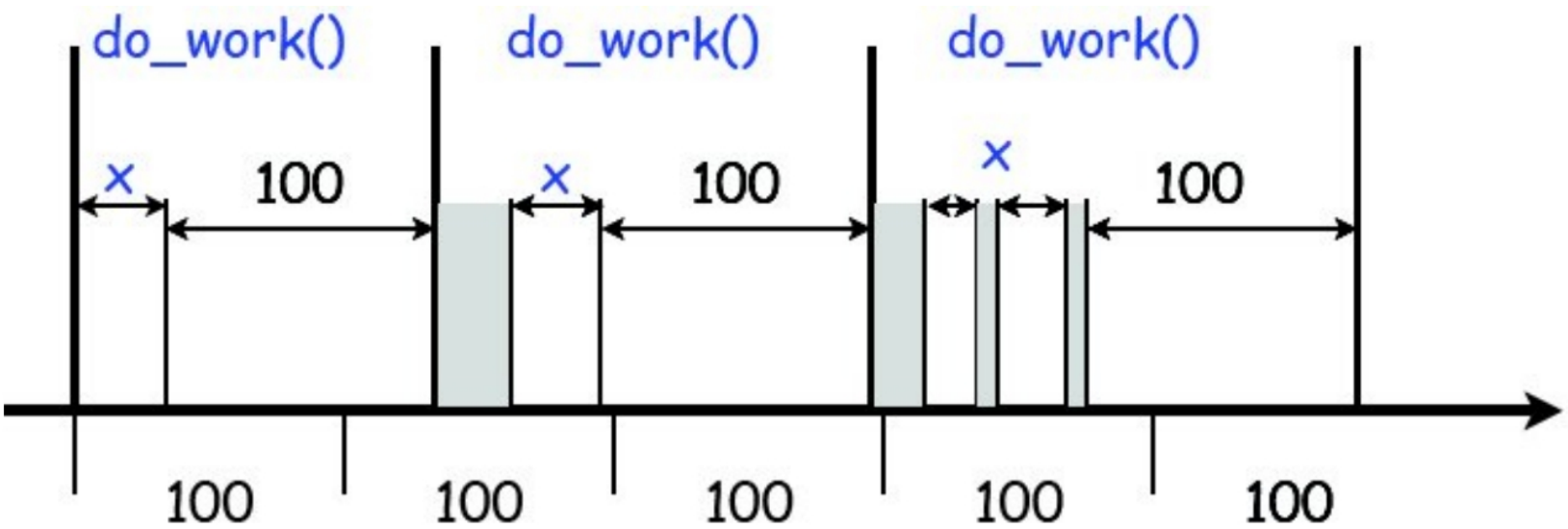
Each turn in the loop will take at least 100 + x milliseconds, where x is the time taken to perform `do_work()`

A Repeated Task (An Attempt)

The goal is to `do work` repeatedly every 100 time units

```
while(1) {
  delay(100);
  do_work();
}
```

Does it work as intended? No, accumulates drift ...



Delay is just lower bound, a delaying process is not guaranteed access to the processor (the delay does not compensate for this)

Eliminating (Part of) The Drift: Timers

- Set an alarm clock, do some work, and then wait for whatever time is left before the alarm rings
- This is done with timers
- Two types of timers
  - one-shot
  - periodic
- Thread is told to wait until the next ring – accumulating drift is eliminated
- Even with timers, drift may still occur, but it does not accumulate (local drift)

Timeouts

Synchronous blocking operations can include timeouts

- Synchronization primitives

  Semaphores, condition variables, locks, etc.

  ... timeout usually generates an error/exception

- Networking and other I/O calls

  E.g. `select()` in POSIX

May also provide an asynchronous timeout signal

- Detect time overruns during execution of periodic and sporadic tasks

Deadline specification and real-time scheduling

Clock driven scheduling trivial to implement via cyclic executive

Other scheduling algorithms need OS and/or language support:

- System calls create, destroy, suspend and resume tasks
- Implement tasks as either threads or processes
  Threads usually more beneficial than processes (with separate address space and memory protection):
  - Processes not always supported by the hardware
  - Processes have longer context switch time
  - Threads can communicate using shared data (fast and more predictable)
- Scheduling support:
  - Preemptive scheduler with multiple priority levels
  - Support for aperiodic tasks (at least background scheduling)
  - Support for sporadic tasks with acceptance tests, etc.

Jobs, Tasks and Threads

- In theory, a system comprises a set of (abstract) *tasks*, each task is a series of *jobs*
  - tasks are typed, have various parameters, react to events, etc.
  - Acceptance test performed before admitting new tasks
- A *thread* (or a process) is the basic unit of work handled by the scheduler
  - Threads are the instantiation of tasks that have been admitted to the system

How to map *tasks* to *threads*?

Periodic Tasks

Real-time tasks defined to execute periodically $T = (\phi, p, e, D)$

It is clearly inefficient if the thread is created and destroyed repeatedly every period

- Some op. systems (funkOS) and programming languages (Real-time Java & Ada) support *periodic threads*
  - the kernel (or VM) reinitializes such a thread and puts it to sleep when the thread completes
  - The kernel releases the thread at the beginning of the next period
  - This provides clean abstraction but needs support from OS
- Thread instantiated once, performs job, sleeps until next period, repeats
  - Lower overhead, but relies on programmer to handle timing
  - Hard to avoid timing drift due to sleep overuns
    (see the discussion of delays earlier in this lecture)
  - Most common approach

Sporadic and Aperiodic Tasks

Events trigger sporadic and aperiodic tasks

- Might be extenal (hardware) interrupts
- Might be signalled by another task

Usual implementation:

- OS executes periodic server thread

  (background server, deferrable server, etc.)
- OS maintains a “server queue” = a list of pointers which give starting addresses of functions to be executed by the server
- Upon the occurrence of an event that releases an aperiodic or sporadic job, the event handler (usually an interrupt routine) inserts a pointer to the corresponding function to the list

Real-Time Programming & RTOS

Real-Time Operating systems

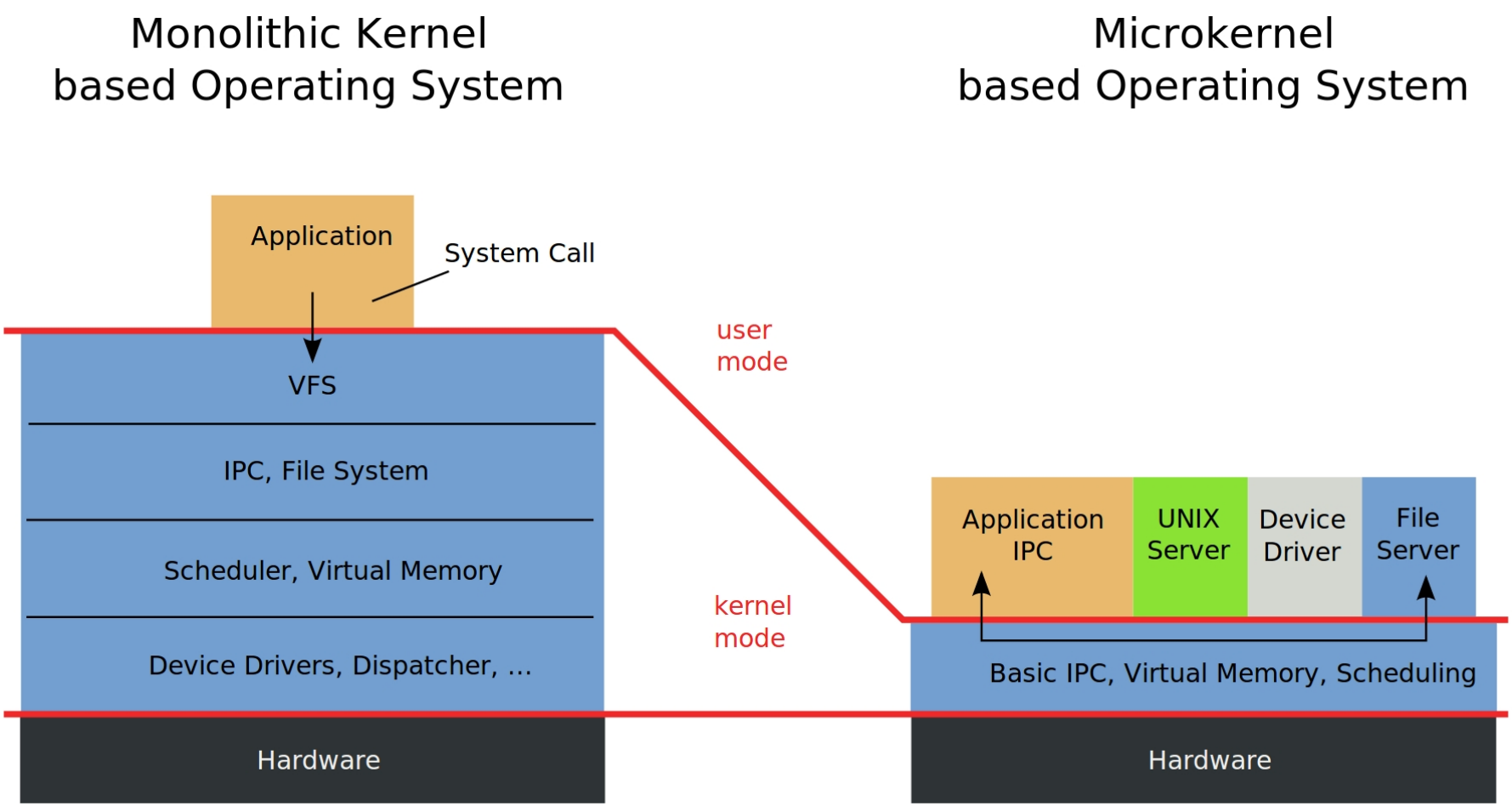
Operating Systems – What You Should Know ...

An operating system is a collection of software that manages computer hardware resources and provides common services for computer programs.

Basic components multi-purpose OS:

- Program execution & process management
  processes (threads), IPC, scheduling, ...
- Memory management
  segmentation, paging, protection ...
- Storage & other I/O management
  files systems, device drivers, ...
- Network management
  network drivers, protocols, ...
- Security
  user IDs, privileges, ...
- User interface
  shell, GUI, ...

Operating Systems – What You Should Know ...



Implementing Real-Time Systems

- Key fact from scheduler theory: *need predictable behavior*
  - Raw performance less critical than consistent and predictable performance; hence focus on scheduling algorithms, schedulability tests
  - Don't want to fairly share resources – be unfair to ensure deadlines met
- Need to run on a wide range of – often custom – hardware
  - Often resource constrained:
    limited memory, CPU, power consumption, size, weight, budget
  - Closed set of applications
    (Do we need a wristwatches to play DVDs?)
  - Strong reliability requirements – may be safety critical
  - How to upgrade software in a car engine? A DVD player?

Implications on Operating Systems

- General purpose operating systems not well suited for real-time
  - Assume plentiful resources, fairly shared amongst untrusted users
  - Serve multiple purposes
  - Exactly opposite of an RTOS!
- Instead want an operating system that is:
  - Small and light on resources
  - Predictable
  - Customisable, modular and extensible
  - Reliable

  ... and that can be *demonstrated* or *proven* to be so

Implications on Operating Systems

- Real-time operating systems typically either *cyclic executive* or *microkernel* designs, rather than a traditional monolithic kernel
  - Limited and well defined functionality
  - Easier to demonstrate correctness
  - Easier to customise
- Provide rich support for concurrency & real-time control
- Expose low-level system details to the applications

  control of scheduling, interaction with hardware devices, ...

Cyclic Executive without Interrupts

- The simplest real-time systems use a “nanokernel” design
  - Provides a minimal time service: scheduled clock pulse with a fixed period
  - No tasking, virtual memory/memory protection etc.
  - Allows implementation of a static cyclic schedule, provided:
    - Tasks can be scheduled in a frame-based manner
    - All interactions with hardware to be done on a polled basis
- Operating system becomes a single task cyclic executive

```
setup timer
c = 0;
while (1) {
          suspend until timer expires
          c++;
          do tasks due every cycle
          if (((c+0) % 2) == 0) do tasks due every 2nd cycle
          if (((c+1) % 3) == 0) {
                    do tasks due every 3rd cycle, with phase 1
          }
          ...
}
```

Microkernel Architecture

- Cyclic executive widely used in low-end embedded devices
  - 8 bit processors with kilobytes of memory
  - Often programmed in (something like) C via cross-compiler, or assembler
  - Simple hardware interactions
  - Fixed, simple, and static task set to execute
  - Clock driven scheduler
- But many real-time embedded systems are more complex, need a sophisticated operating system with priority scheduling
- Common approach: a *microkernel* with priority scheduler

  Configurable and robust, since architected around interactions between cooperating system servers, rather than a monolithic kernel with ad-hoc interactions

Microkernel Architecture

- A microkernel RTOS typically provides:
  - Timing services, interrupt handling, support for hardware interaction
  - Task management, scheduling
  - Messaging, signals and events
  - Synchronization and locking
  - Memory management (and sometimes also protection)

Latency

(Some) sources of hard to predict latency caused by the system:

- Interrupts

  see next slide
- System calls

  RTOS should characterise WCET; kernel should be preemptable
- Memory management: paging

  avoid, either use segmentation with a fixed memory management scheme, or memory locking
- Caches

  may introduce non-determinism; there are techniques for computing WCET with processor caches
- DMA

  competes with processor for the memory bus, hard to predict who wins

Interrupts

The amount of time required to handle interrupt varies

Thus in most OS, interrupt handling is divided into two steps

- Immediate interrupt service

  very short; invokes a scheduled interrupt handling routine
- Scheduled interrupt service

  preemptable, scheduled as an ordinary job at a suitable priority

Immediate Interrupt Service

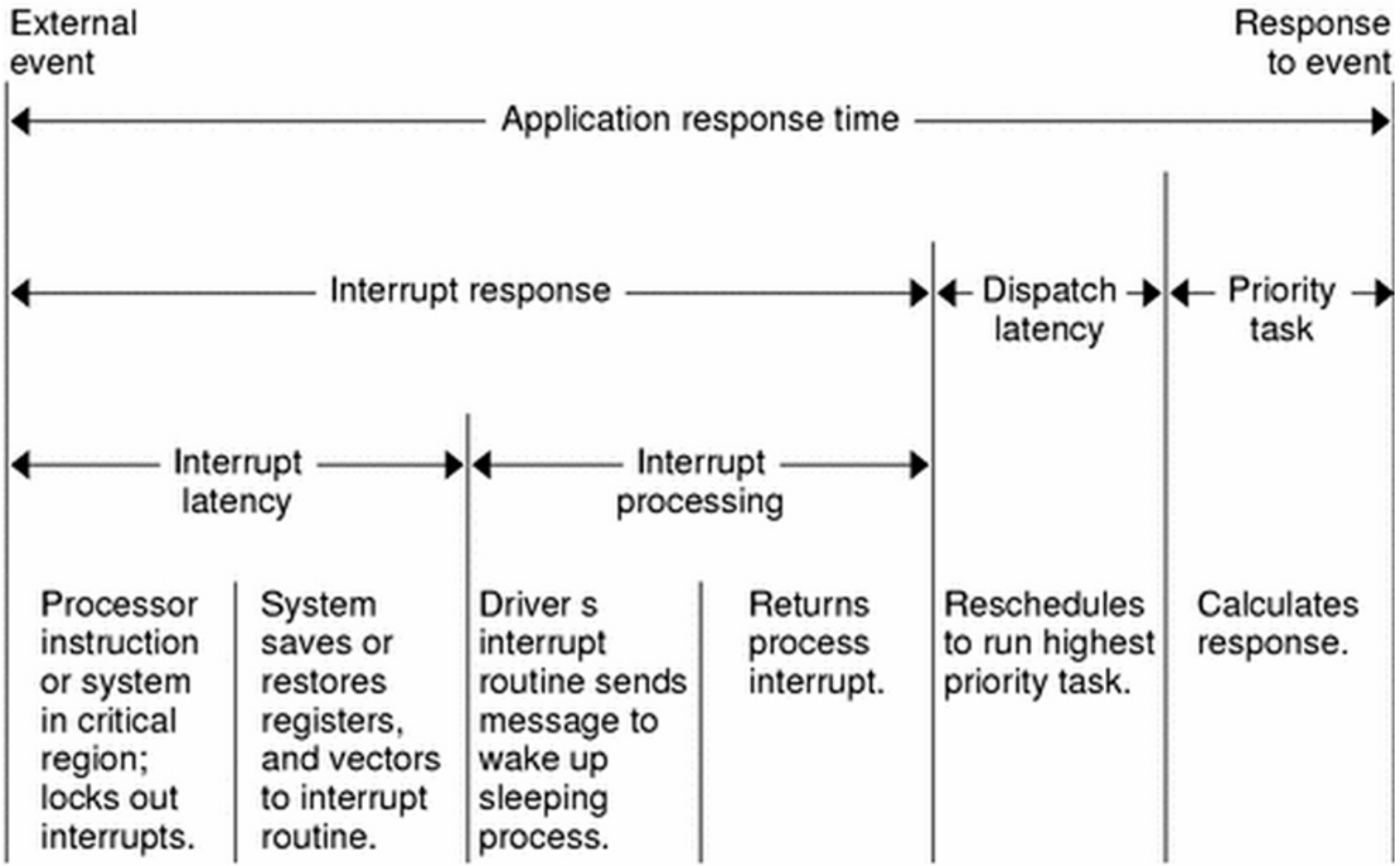
Interrupt latency is the time between interrupt request and execution of the first instruction of the interrupt service routine

The total delay caused by interrupt is the sum of the following factors:

- the time the processor takes to complete the current instruction, do the necessary chores (flush pipeline and read the interrupt vector), and jump to the trap handler and interrupt dispatcher
- the time the kernel takes to disable interrupts
- the time required to complete the immediate interrupt service routines with higher-priority interrupts (if any) that occurred simultaneously with this one
- the time the kernel takes to save the context of the interrupted thread, identify the interrupting device, and get the starting address of the interrupt service routine
- the time the kernel takes to start the interrupt service routine

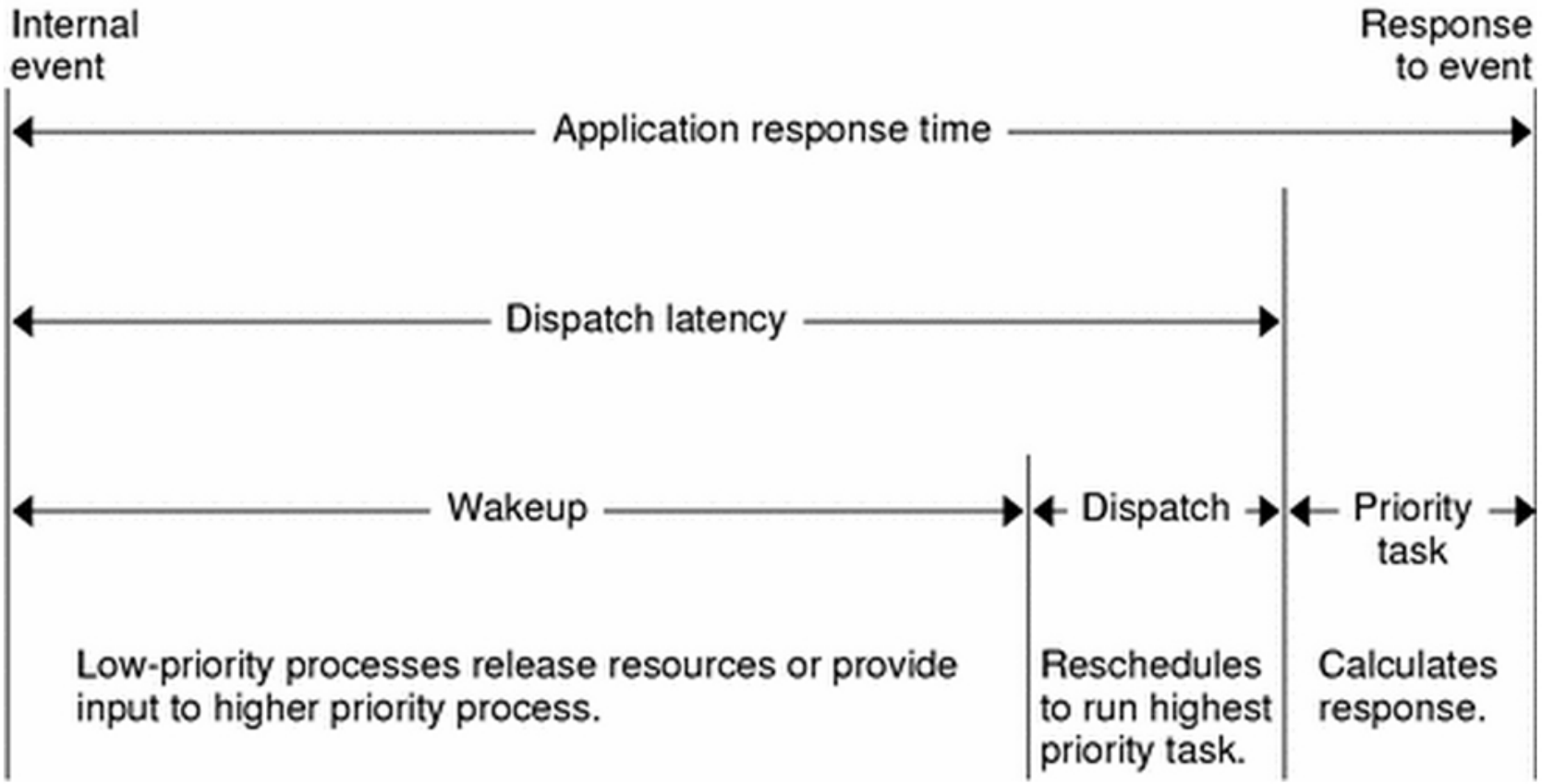
Event Latency

External event latency in an event driven system:



Event Latency

Internal event latency in an event driven system:



Hard real-time OS should have a guaranteed upper bound on event latency

Example RTOS: FreeRTOS

- RTOS for embedded devices (currently ported to 31 microcontrollers)
- Distributed under GPL
- Written in C, kernel consists of 3+1 C source files
  (approx. 9000 lines of code including comments)
- Largely configurable

Example RTOS: FreeRTOS

- The OS is (more or less) a library of object modules;
  the application and OS modules are linked together in the resulting executable image
- Prioritized scheduling of tasks
  - tasks correspond to threads (share the same address space; have their own execution stacks)
  - highest priority executes; same priority ⇒ round robin
  - implicit *idle task* executing when no other task executes ⇒ may be assigned functionality of a background server
- Synchronization using semaphores
- Communication using message queues
- Memory management
  - no memory protection in basic version (can be extended)
  - various implementations of memory management
    memory can/cannot be freed after allocation, best fit vs combination of adjacent memory block into a single one

That's (almost) all ....

Example RTOS: FreeRTOS

Tiny memory requirements: e.g. IAR STR71x ARM7 port, full optimisation, minimum configuration, four priorities $\Rightarrow$

- size of the scheduler = 236 bytes
- each queue adds 76 bytes + storage area
- each task 64 bytes + the stack size

Real-Time Programming & RTOS

Real-Time Programming Languages

Brief Overview

C and POSIX

IEEE 1003 POSIX

- "Portable Operating System Interface"
- Defines a subset of Unix functionality, various (optional) extensions added to support real-time scheduling, signals, message queues, etc.
- Widely implemented:
  - Unix variants and Linux
  - Dedicated real-time operating systems
  - Limited support in Windows

Several POSIX standards for real-time scheduling

- POSIX 1003.1b ("real-time extensions")
- POSIX 1003.1c ("pthreads")
- POSIX 1003.1d ("additional real-time extensions")
- Support a sub-set of scheduler features we have discussed

POSIX Scheduling API (Processes)

```
#include <unistd.h>
#include <sched.h>

struct sched_param {
        int             sched_priority;
        int             sched_ss_low_priority;
        struct timespec sched_ss_repl_period;
        struct timespec sched_ss_init_budget;
};

int sched_setscheduler(pid_t pid, int policy, struct sched_param *p);
int sched_getscheduler(pid_t pid);
int sched_getparam(pid_t pid, struct sched_param *sp);
int sched_setparam(pid_t pid, struct sched_param *sp);

int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);

int sched_rr_get_interval(pid_t pid, struct timespec *t);

int sched_yield(void);
```

POSIX Scheduling API (Threads)

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

int pthread_attr_init(pthread_attr_t *attr);

int pthread_attr_getschedpolicy(pthread_attr_t *attr, int policy);
int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);

int pthread_attr_getschedparam(pthread_attr_t *attr, struct sched_param *p);
int pthread_attr_setschedparam(pthread_attr_t *attr, struct sched_param *p);

int pthread_create(pthread_t      *thread,
                   pthread_attr_t *attr,
                   void *(*thread_func)(void*),
                   void   *thread_arg);
int pthread_exit(void *retval);
int pthread_join(pthread_t thread, void **retval);
```

Thread scheduling API mirrors process scheduling API

same scheduling policies, priorities, etc.

Threads: Example I

```
#include <pthread.h>

pthread_t id;
void *fun(void *arg) {
  // Some code sequence
}

main() {
  pthread_create(&id, NULL, fun, NULL);
  // Some other code sequence
}
```

Threads: Example II

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS     5

void *PrintHello(void *threadid)
{
   printf("\n%d: Hello World!\n", threadid);
   pthread_exit(NULL);
}

int main (int argc, char *argv[])
{
   pthread_t threads[NUM_THREADS];
   int rc, t;
   for(t=0; t<NUM_THREADS; t++){
      printf("Creating thread %d\n", t);
      rc = pthread_create(&threads[t], NULL, PrintHello, (void *)t);
      if (rc){
         printf("ERROR; return code from pthread_create() is %d\n", rc);
         exit(-1);
      }
   }
   pthread_exit(NULL);
}
```

POSIX: Synchronization

Mutexes = variables that can be locked by processes (threads)

- `pthread_mutex_init(mutex,attr)`
- `pthread_mutex_lock(mutex)` – attempt to lock a mutex, if the mutex is already locked, this call blocks the thread (may implement a priority inheritance protocol)
- `pthread_mutex_trylock(mutex)` – if the mutex is locked, returns immediately with "busy" error code
- `pthread_mutex_unlock(mutex)`

POSIX: Communication – Signals

Signal is an *asynchronous* notification sent to a process (or a thread) in order to notify it of an event that occurred.

When a signal is sent, the operating system interrupts the target process's normal flow of execution to deliver the signal.

Execution can be interrupted during any non-atomic instruction.

Moreover,

- Signal can be sent to a process by executing `kill(pid,sig)` where `pid` is the process number (0 means self)
- Signals are also generated by dividing with zero, addresing outside your address space, etc.
- Each thread can block incoming signals on a per-signal basis, define signal handlers for each signal it might receive, and queue signals
- No data transfer
- Can be used to handle exceptions

POSIX: Communication – List of (Some) Signals

SIGABRT	Abnormal termination signal caused by the abort() function.
SIGALRM	The timer has timed-out.
SIGFPE	Arithmetic exception, such as overflow or division by zero.
SIGHUP	Hangup detected on controlling terminal or death of a controlling process.
SIGILL	Illegal instruction indicating a program error.
SIGINT	Interrupt special character typed on controlling keyboard (Ctrl-C).
SIGKILL	Termination signal. This signal cannot be caught or ignored.
SIGPIPE	Write to a pipe with no readers.
SIGQUIT	Quit special character typed on controlling keyboard.
SIGSEGV	Invalid memory reference. Like SIGILL, portable programs should not intentionally generate invalid memory references.
SIGTERM	Termination signal.
SIGUSR1	Application-defined signal 1.
SIGUSR2	Application-defined signal 2.
SIGCHLD	Child process terminated or stopped.
SIGCONT	Continue the process if it is currently stopped; otherwise, ignore the signal.

POSIX: Communication – Signal Handling

- same basic idea as for real interrupt-handling; a handler for a signal gets called "spontaneously", just as if the interrupted code had made the call itself
- like an interrupt handler ignores what process is running, a signal handler ignores what thread is running
- difference: signals are not delivered until the receiving process is actually running
- internally generated signals the receiving process is already running per definition

POSIX: Message Passing

- POSIX supports asynchronous, indirect message passing through the notion of message queues
- A message queue can have many readers and many writers
- Intended for communication between processes (not threads)
- Message queues have attributes which indicate their maximum size, the size of each message, the number of messages currently queued etc.

- `mq_open` (also creates queue), `mq_close`, `mq_send`, `mq_receive`
- Data is read/written from/to a character buffer.
- If the buffer is full or empty, the sending/receiving process is blocked unless the attribute `O_NONBLOCK` has been set for the queue (in which case an error return is given)
- If senders and receivers are waiting when a message queue becomes unblocked, it is not specified which one is woken up unless the priority scheduling option is specified

... for more see `sys/ipc.h`, `sys/msg.h`, `mqueue.h`, ...

POSIX: Real-Time Support

Getting Time

- `time()` = seconds since Jan 1 1970
- `gettimeofday()` = seconds + nanoseconds since Jan 1 1970
- `tm` = structure for holding human readable time

```
struct tm {
  int   tm_sec;     // seconds (0 - 60)
  int   tm_min;     // minutes (0 - 59)
  int   tm_hour;    // hours (0 - 23)
  int   tm_mday;    // day of month (1 - 31)
  int   tm_mon;     // month of year (0 - 11)
  int   tm_year;    // year - 1900
  int   tm_wday;    // day of week (Sunday = 0)
  int   tm_yday;    // day of year (0 - 365)
  int   tm_isdst;   // is summer time in effect?
  char *tm_zone;    // timezone name
  long  tm_gmtoff;  // offset from UTC
};
struct tm localtime(time_t t);
time_t    mktime(struct tm *t);
```

- POSIX requires at least one clock of minimum resolution 50Hz (20ms)

POSIX: High Resolution Time & Timers

High resolution clock. Known clock resolution.

```
#include <sys/time.h>

struct timespec {
        time_t   tv_sec;
        long     tv_nsec;
};
int clock_gettime(CLOCK_REALTIME, struct timespec *t);
int clock_getres (CLOCK_REALTIME, struct timespec *r);
```

Simple waiting: `sleep`, or

```
int nanosleep(struct timespec *delay, struct timespec *remaining);
```

Sleep for the interval specified. May return earlier due to signal (in which case `remaining` gives the remaining delay).

Accuracy of the delay not known (and not necessarily correlated to `clock_getres()` value)

POSIX: Timers

- type `timer_t`; can be set:
  - relative/absolute time
  - single alarm time and an optional repetition period
- timer “rings” by sending a signal

```
int timer_create(clockid_t clockid, struct sigevent *sevp,
                 timer_t *timerid);


int timer_settime(timer_t timerid, int flags,
                  const struct itimerspec *new_value,
                  struct itimerspec * old_value);
```

where

```
struct itimerspec {
            struct timespec it_interval;  /* Timer interval */
            struct timespec it_value;     /* Initial expiration */
        };
```

POSIX Scheduling API

- Four scheduling policies:
  - `SCHED_FIFO` = Fixed priority, pre-emptive, FIFO scheduler
  - `SCHED_RR` = Fixed priority, pre-emptive, round robin scheduler
  - `SCHED_SPORADIC` = Sporadic server
  - `SCHED_OTHER` = Unspecified (often the default time-sharing scheduler)
- A process can `sched_yield()` or otherwise block at any time
- POSIX 1003.1b provides (largely) fixed priority scheduling
  - Priority can be changed using `sched_set_param()`, but this is high overhead and is intended for reconfiguration rather than for dynamic scheduling
  - No direct support for dynamic priority algorithms (e.g. EDF)
- Limited set of priorities:
  - Use `sched_get_priority_min()`, `sched_get_priority_max()` to determine the range
  - Guarantees at least 32 priority levels

Using POSIX Scheduling: Rate Monotonic

Rate monotonic and deadline monotonic schedules can be naturally implemented using POSIX primitives

1. Assign priorities to tasks in the usual way for RM/DM
2. Query the range of allowed system priorities `sched_get_priority_min()` and `sched_get_priority_max()`
3. Map task set onto system priorities

   Care needs to be taken if there are large numbers of tasks, since some systems only support a few priority levels
4. Start tasks using assigned priorities and `SCHED_FIFO`

There is no explicit support for indicating deadlines, periods

EDF scheduling not supported by POSIX

Using POSIX Scheduling: Sporadic Server

POSIX 1003.1d defines a hybrid sporadic/background server

```
struct sched_param {
  int              sched_priority;
  int              sched_ss_low_priority;
  struct timespec sched_ss_repl_period;
  struct timespec sched_ss_init_budget;
};
```

When server has budget, runs at `sched_priority`, otherwise runs as a background server at `sched_ss_low_priority`

Set `sched_ss_low_priority` to be lower priority than real-time tasks, but possibly higher than other non-real-time tasks in the system

Also defines the replenishment period and the initial budget after replenishment

POSIX-compliant RTOS

Examples of POSIX-compliant implementations:

- commercial:
  - VxWorks
  - QNX
  - OSE
- Linux-related:
  - RTLINUX
  - RTAI

Java

- object-oriented programming language
- developed by Sun Microsystems in the early 1990s
- compiled to bytecode (for a *virtual machine*), which is compiled to native machine code at runtime
- syntax of Java is largely derived from C/C++

Concurrency: Threads

- predefined class `java.lang.Thread` – provides the mechanism by which threads (processes) are created
- to avoid all threads having to be child classes of Thread, it also uses a standard interface:

```
public interface Runnable {
    public abstract void run();
}
```

- any class which wishes to express concurrent execution must implement this interface and provide the `run()` method

Threads: Creation & Termination

Creation:

- dynamic thread creation, arbitrary data to be passed as parameters
- thread hierarchies and thread groups can be created

Termination:

- one thread can wait for another thread (the target) to terminate by issuing the join method call on the target's thread object
- the `isAlive` method allows a thread to determine if the target thread has terminated
- garbage collection cleans up objects which can no longer be accessed
- main program terminates when all its user threads have terminated

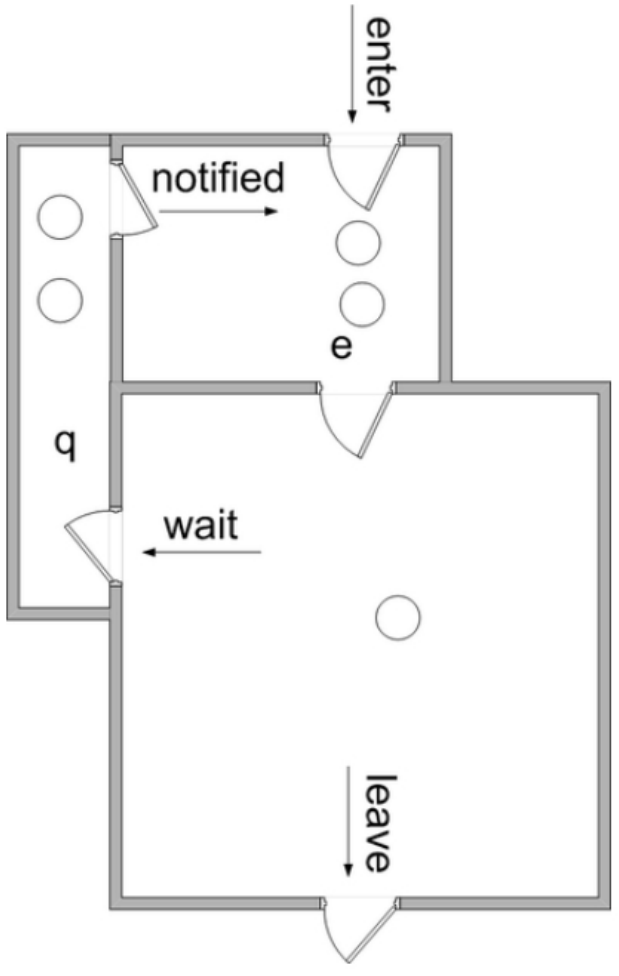
Synchronized Methods

- monitors can be implemented in the context of classes and objects
- lock associated with each object; lock cannot be accessed directly by the application but is affected by
  - the method modifier `synchronized`
  - block synchronization
- `synchronized` method – access to the method can only proceed once the lock associated with the object has been obtained
- non-synchronized methods do not require the lock, can be called at any time

Waiting and Notifying

- `wait()` always blocks the calling thread and releases the lock associated with the object
- `notify()` wakes up one waiting thread

  which thread is woken is not defined
- `notifyAll()` wakes up all waiting threads
- if no thread is waiting, then `notify()` and `notifyAll()` have no effect



Real-Time Java

- Standard Java is not enough to handle real-time constraints
- Java (and JVM) lacks semantic for standard real-time programming techniques.
- Embedded Java Specification was there, but merely a subset of standard Java API.
- There is a gap for a language real-time capable and equipped with all Java's powerful advantages.

- IBM, Sun and other partners formed Real-time for Java Expert Group sponsored by NIST in 1998
- It came up with Real-Time Specification for Java (RTSJ) to fill this gap for real-time systems
- RTSJ proposed seven areas of enhancements to the standard Java

RTSJ – Areas of Enhancement

1. Thread scheduling and dispatching
   see the next slides
2. Memory management
   immortal memory (no garbage collection), threads not preemptable by garbage collector
3. Synchronization and resource sharing
   priority inheritance and ceiling protocols
4. Asynchronous event handling, asynchronous transfer of control, asynchronous thread termination
   reaction to OS-level signals (POSIX), hardware interrupts and custom events defined and fired by the application
5. Physical memory access

The resulting real-time extension needs a modified Java virtual machine due to changes to memory model, garbage collector and thread scheduling

Real-Time Java: Time

- `java.lang.System.currentTimeMilis` returns the number of milliseconds since Jan 1 1970
- Real Time Java adds real time clocks with high resolution time types

Timers

- one shot timers (`javax.realtime.OneShotTimer`)
- periodic timers (`javax.realtime.PeriodicTimer`)

Constructor:

`Timer(HighResolutionTime t, Clock c, AsyncEventHandler handler)`

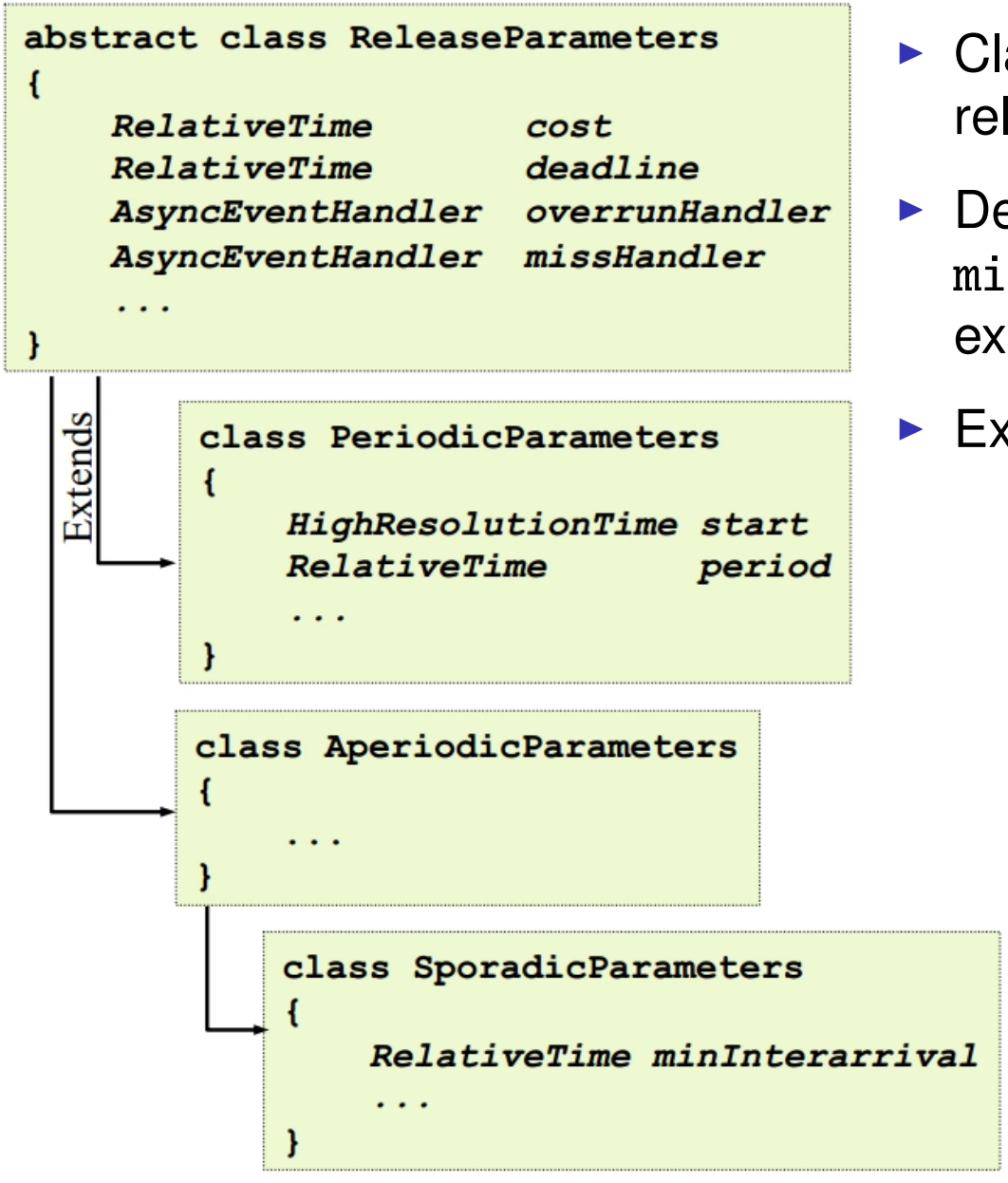
... create a timer that fires at time `t`, according to `Clock c` and is handled by the specified handler.

Real-Time Thread Scheduling

- Extends Java with a `schedulable` interface and `RealtimeThread` class, and numerous supporting libraries

  $\Rightarrow$ Allows definition of timing and scheduling parameters, and memory requirements of threads
- Abstract `Scheduler` and `SchedulingParameters` classes defined
  - Allows a range of schedulers to be developed
    - Current standards only allow system-defined schedulers; cannot write a new scheduler without modifying the JVM
  - Current standards provide a pre-emptive fixed priority scheduler (`PriorityScheduler` class)
    - Allows monitoring of execution times; missed deadlines; CPU budgets
    - Allows thread priority to be changed programatically
    - Limited support for acceptance tests (`isFeasible()`)

Real-Time Thread Scheduling



- Class hierarchy to express release timing parameters
- Deadline monitoring: `missHandler` if deadline exceeded
- Execution time monitoring:
  - `cost` = needed CPU time
  - `overrunHandler` if execution time budget exceeded

Real-Time Thread Scheduling

```
class RealtimeThread extends java.lang.Thread
{
    // ...adds additional constructors to specify
    // ReleaseParameters and SchedulingParameters
    ...

    // ...adds additional methods:
    public void    setScheduler(Scheduler s);
    public void    schedulePeriodic();
    public boolean waitForNextPeriod();
    ...
}
```

- The `RealtimeThread` class extends `Thread` with extra methods and parameters
  - Direct support for periodic threads
    - `run()` method will be a loop ending in a `waitForNextPeriod()` call
    - ... i.e. does not have to be implemented using sleep as e.g. with POSIX API

Ada

- designed for United States Department of Defense during 1977-1983
- targeted at embedded and real-time systems
- Ada 95 revision
- used in critical systems (avionics, weapon systems, spacecrafts)
- free compiler: `gnat`

Ada Lovelace (1815-1852)

Main Principles

- structured, statically typed imperative computer programming language
- strong typing
- modularity mechanisms (packages)
- run-time checking

  e.g. range checks and overflow checks of subtypes
- parallel processing (tasks)
- exception handling
- object-oriented programming (Ada95)

Concurrency: Tasks

- task = the unit of concurrency
- explicitly declared (no fork/join statement, cobegin, ...)
- tasks may be declared at any program level
- created implicitly upon entry to the scope of their declaration or via the action of an allocator

Tasks: Interaction

- communication and synchronization via a variety of mechanisms:
  - rendezvous (a form of synchronised message passing)
  - protected units (a form of monitor)
  - shared variables
- support for hierarchies of tasks (one task waits before the other ends etc.)

- remote invocation with direct asymmetric naming
- one task defines an entry and then, within its body, accepts any incoming call (`accept` statement)
- a randezvous occurs when one task calls an entry in another task
- selective waiting allows a process to wait for more than one message

Time

- access to clock:
  - package `Calendar`
  - abstract data type `Time`
  - function `Clock` for reading time
  - data type `Duration` predefined fixed point real for time calculations
  - conversion utilities (to human readable units)
- waiting: `delay`, `delay until` statements

Example

```
task Ticket_Agent is
  entry Registration(...);
end Ticket_Agent;

task body Ticket_Agent is
  -- declarations
  Shop_Open : Boolean := True;
begin
  while Shop_Open loop
    select
      accept Registration(...) do
         -- log details
      end Registration;
    or
      delay until Closing_Time;
      Shop_Open := False;
    end select;
    -- process registrations
  end loop;
end Ticket_Agent;
```