Operating System & POSIX

What is an Operating System?

- It is an *extended machine*
  - Hides the messy details that must be performed
  - Presents user with a virtualized and simplified abstraction of the machine, easier to use

- It is a *resource manager*
  - Each program gets time with the resource
  - Each program gets space on the resource

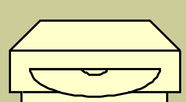
What is an operating system and why do I need one?

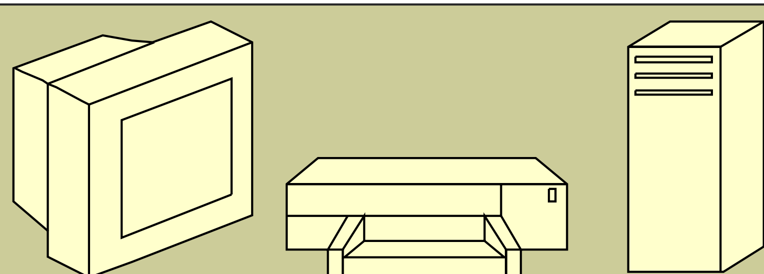
Application Software

Firefox	Second Life	Yahoo Chat	GMail

- A clean way to allow applications to use these resources!

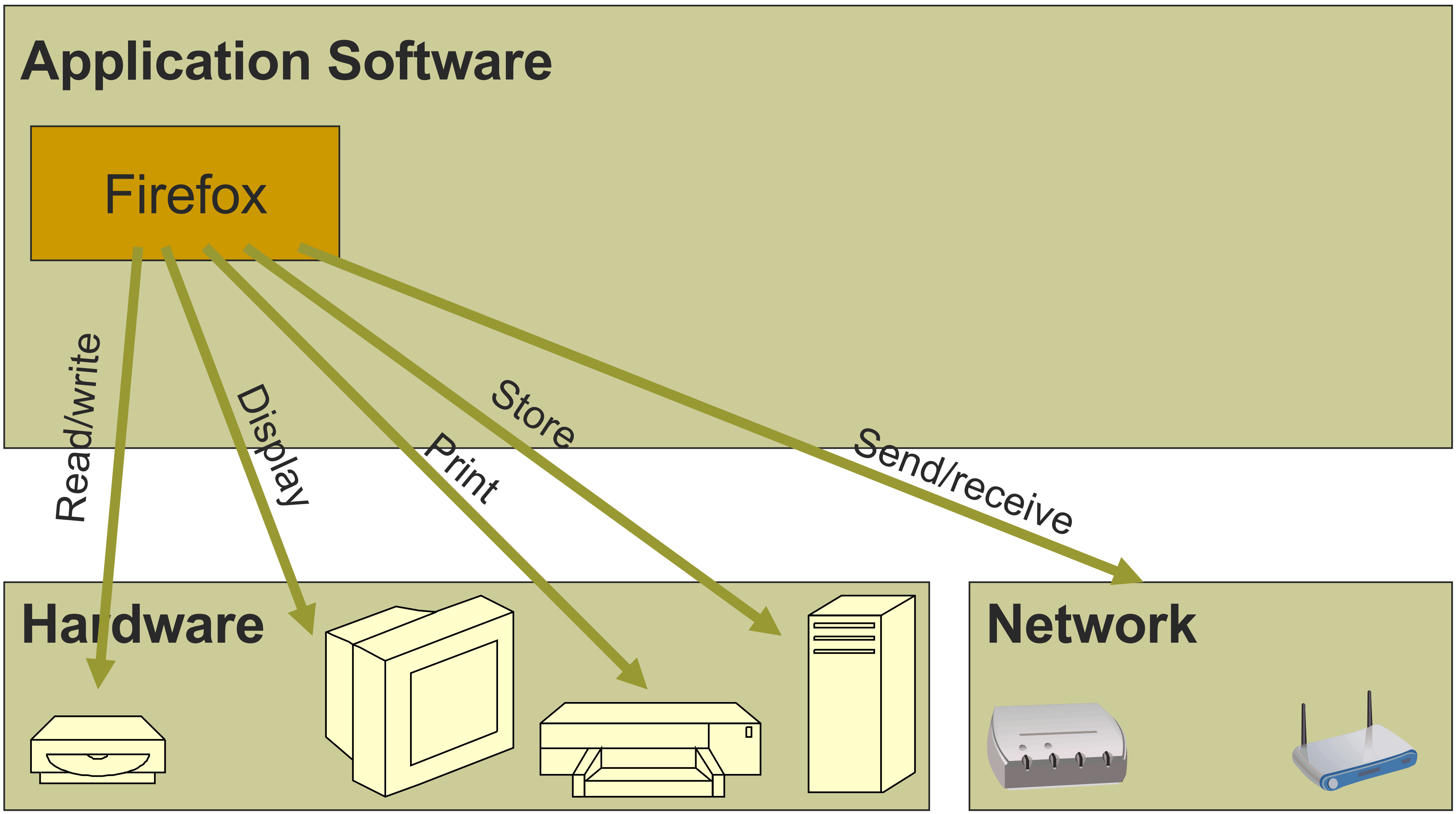
Hardware



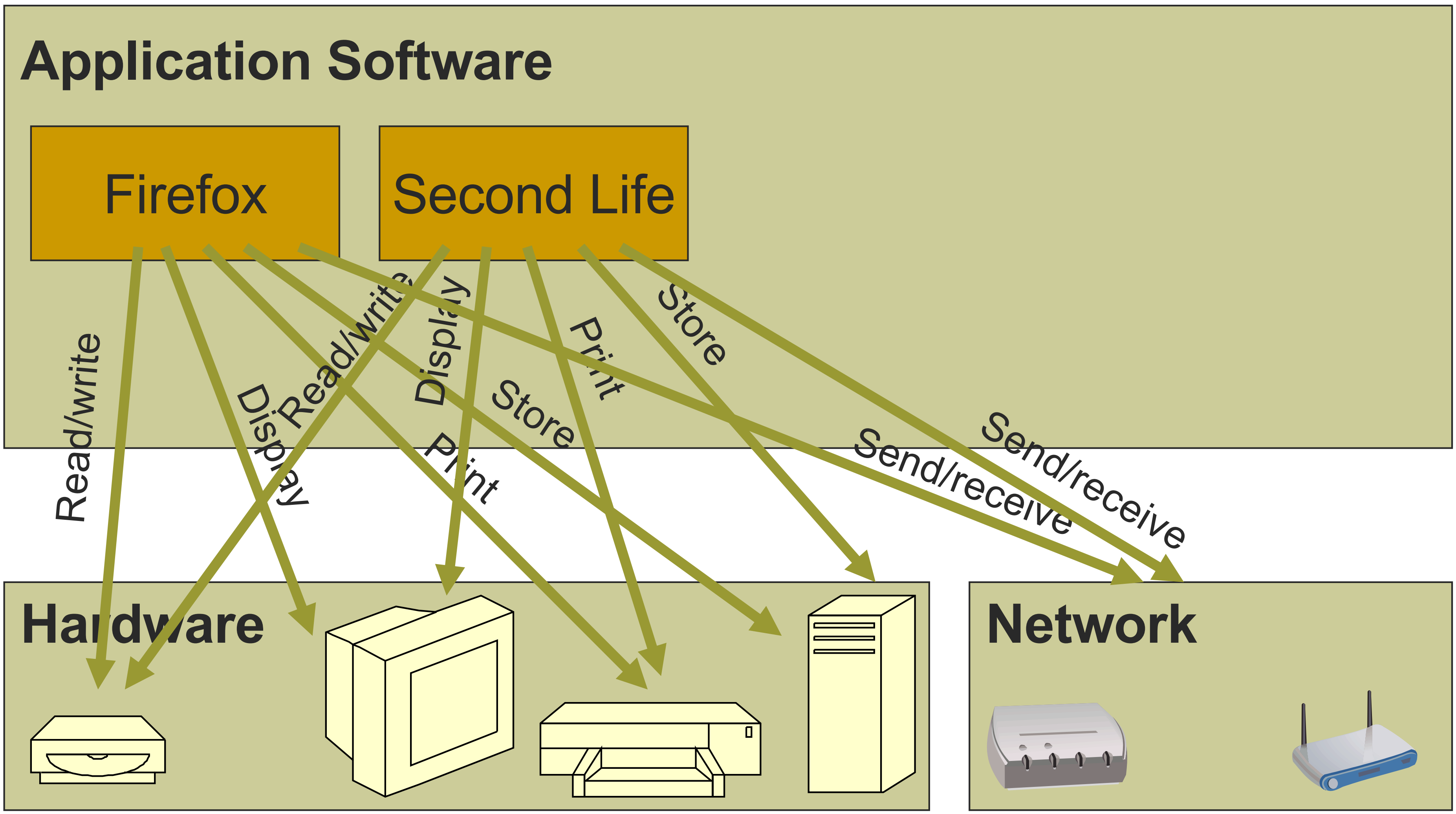


Network

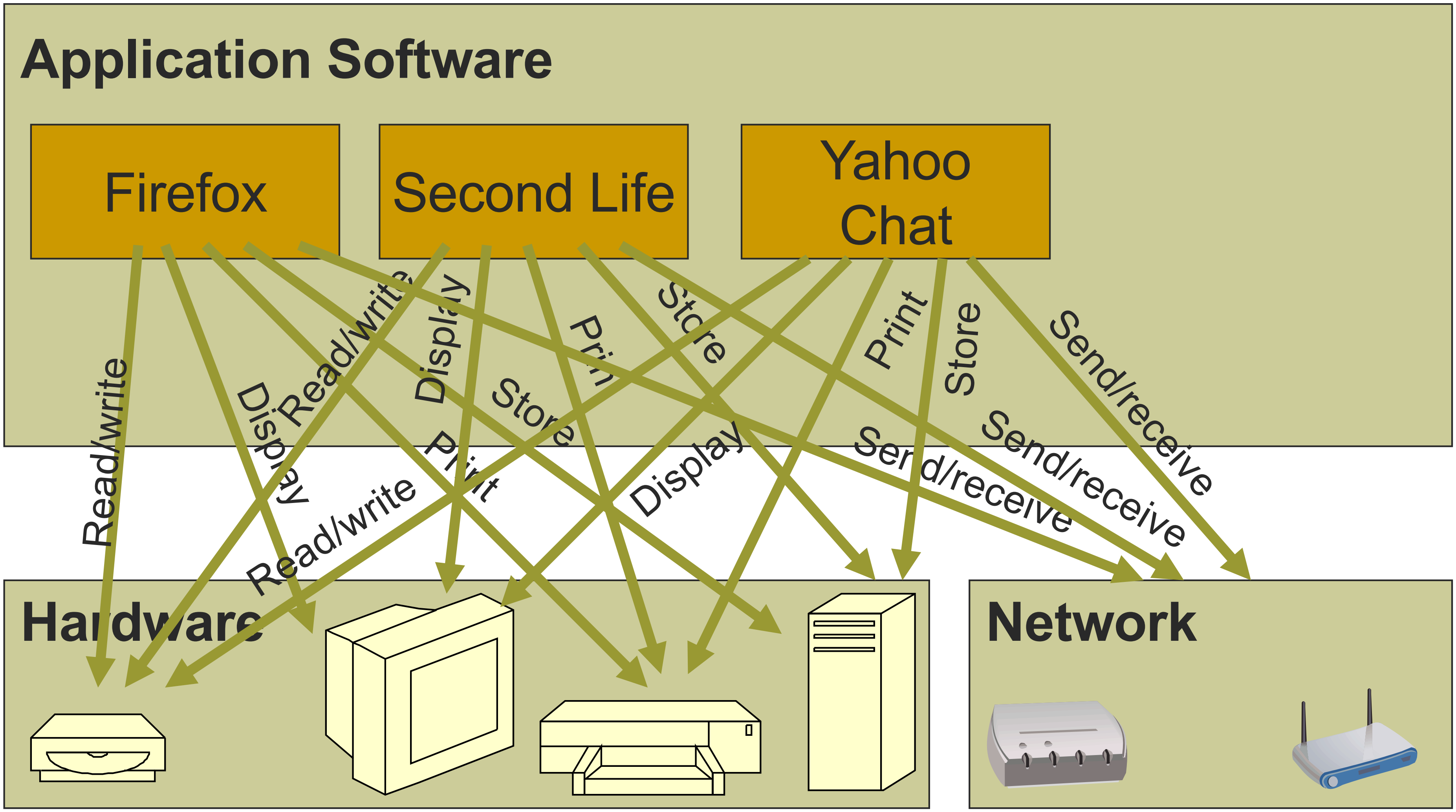
Application Requirements



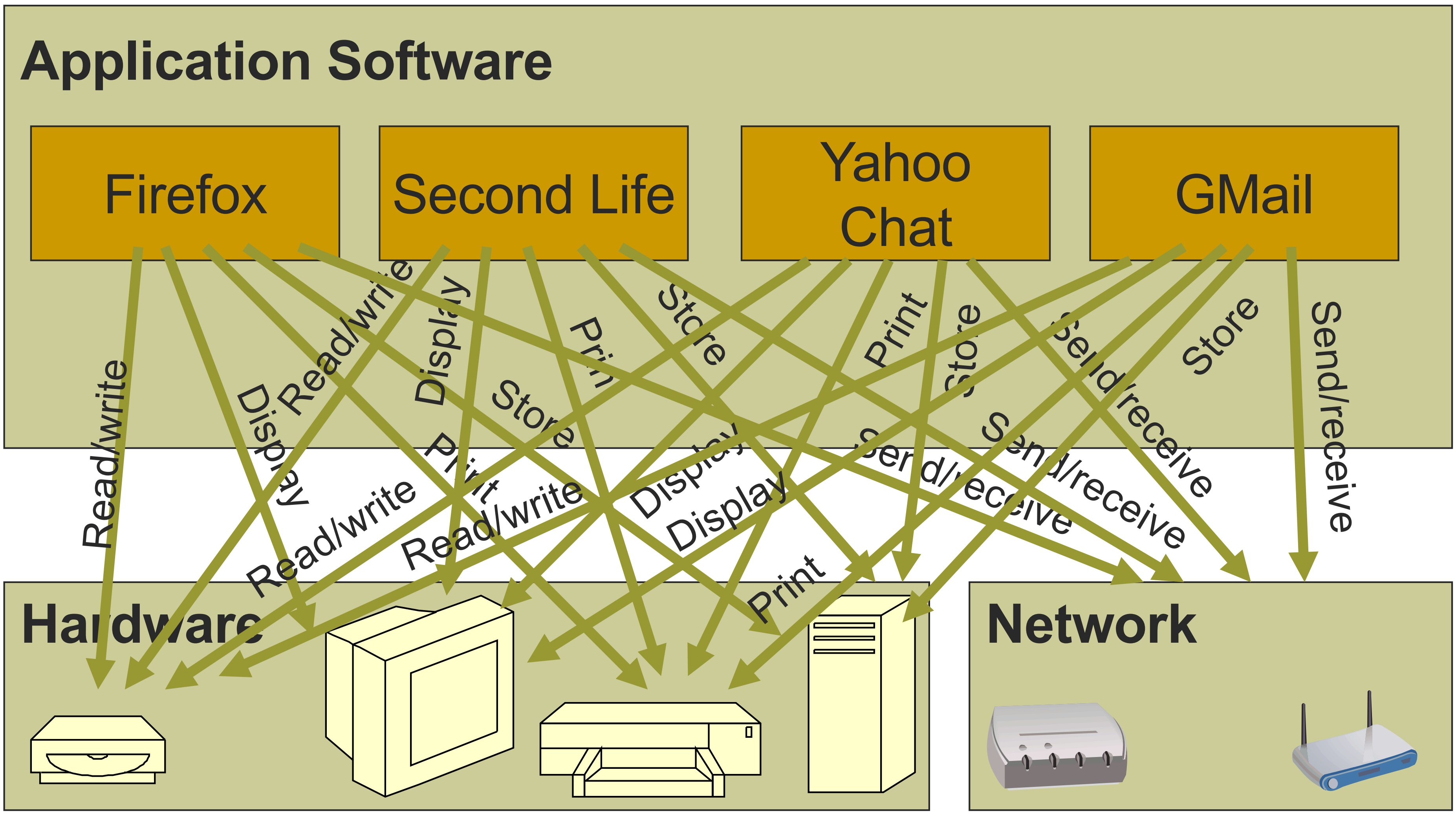
Two Applications?



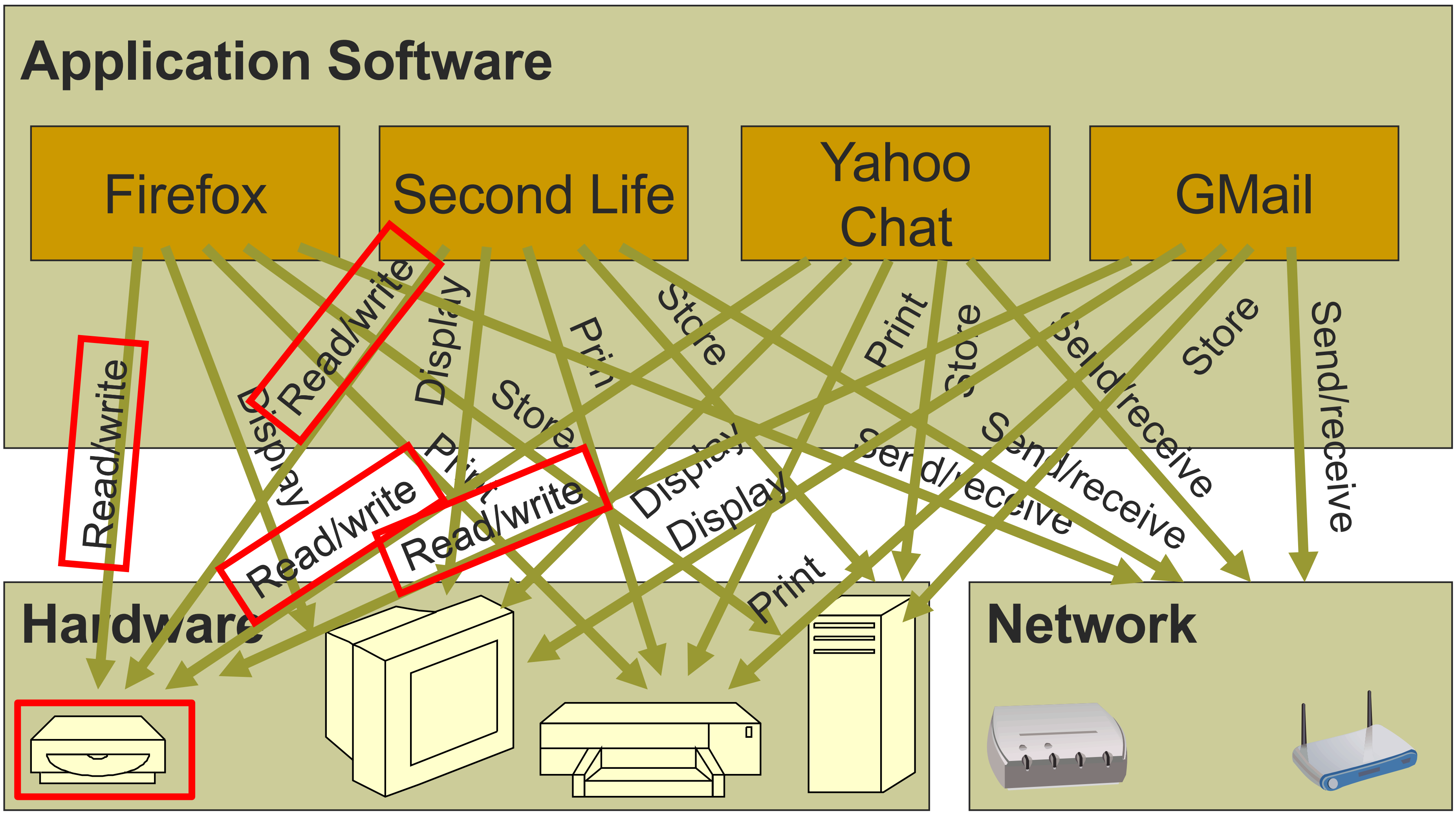
Managing More Applications?



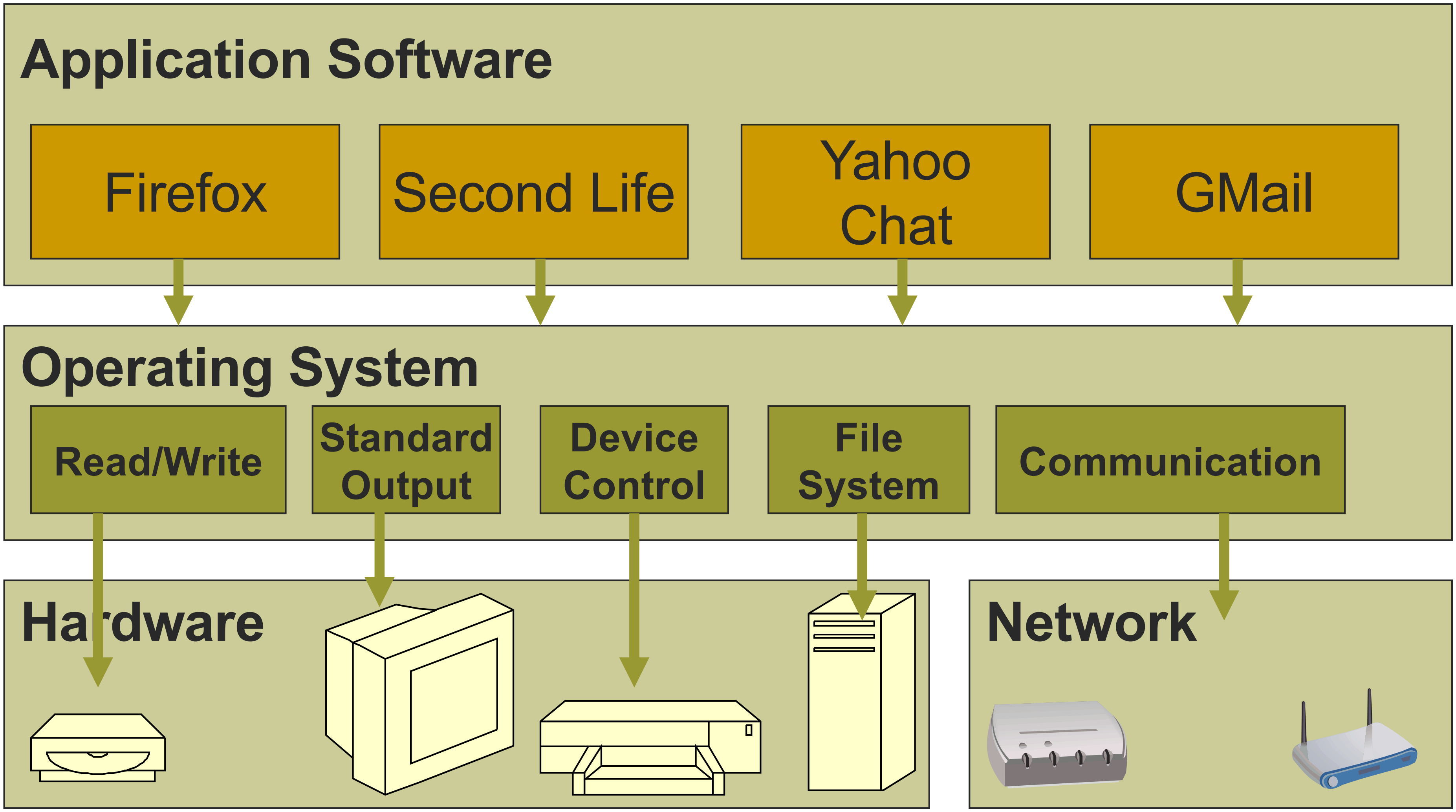
We need help!


Application Software
Firefox
Second Life
Yahoo Chat
GMail
Read/write
Display
Read/write
Display
Store
Print
Store
Print
Store
Send/receive
Store
Send/receive
Send/receive
Send/receive
Print
Read/write
Read/write
Display
Display
Print
Hardware
Network

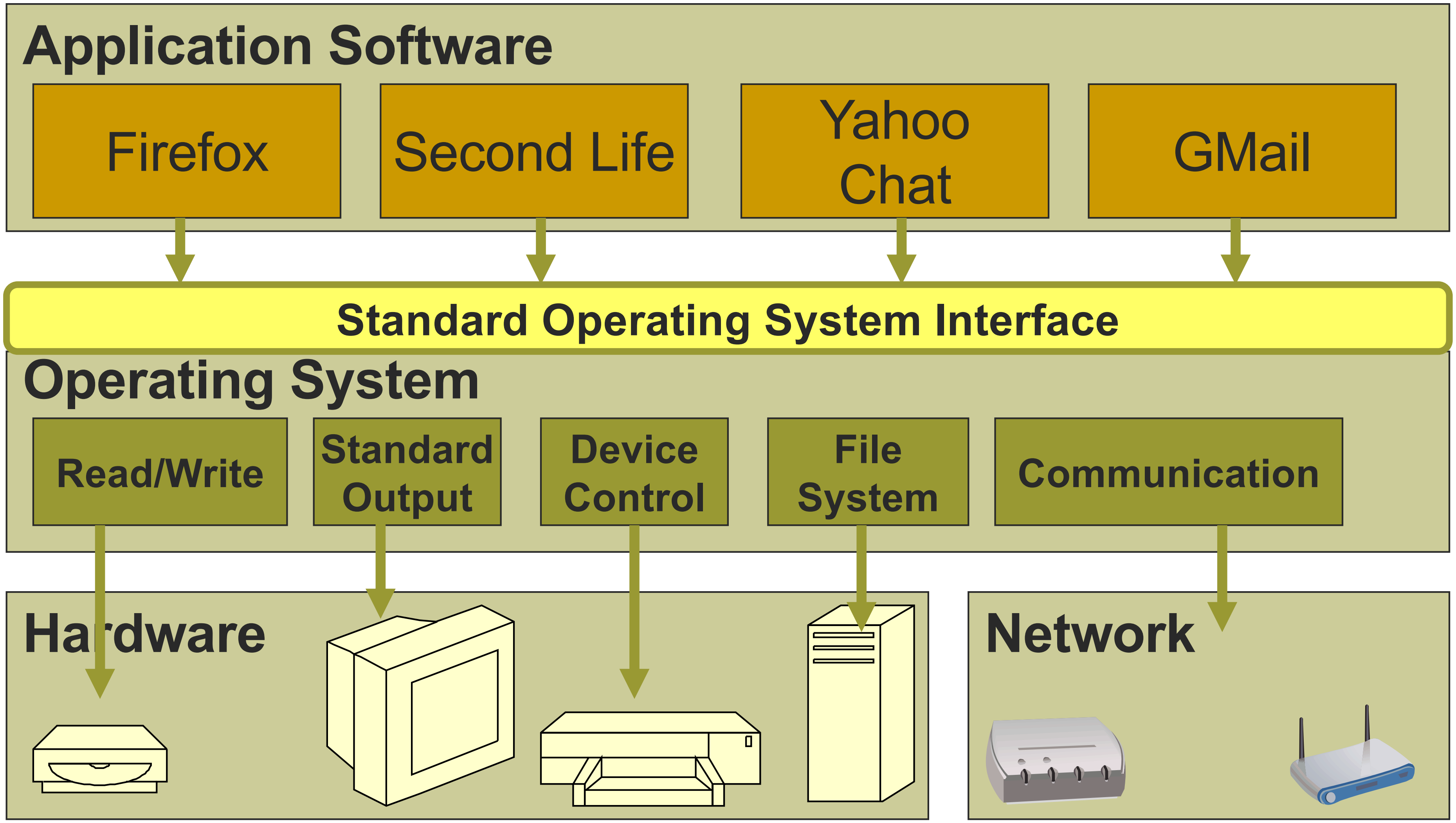
Approach: Find Common Functions



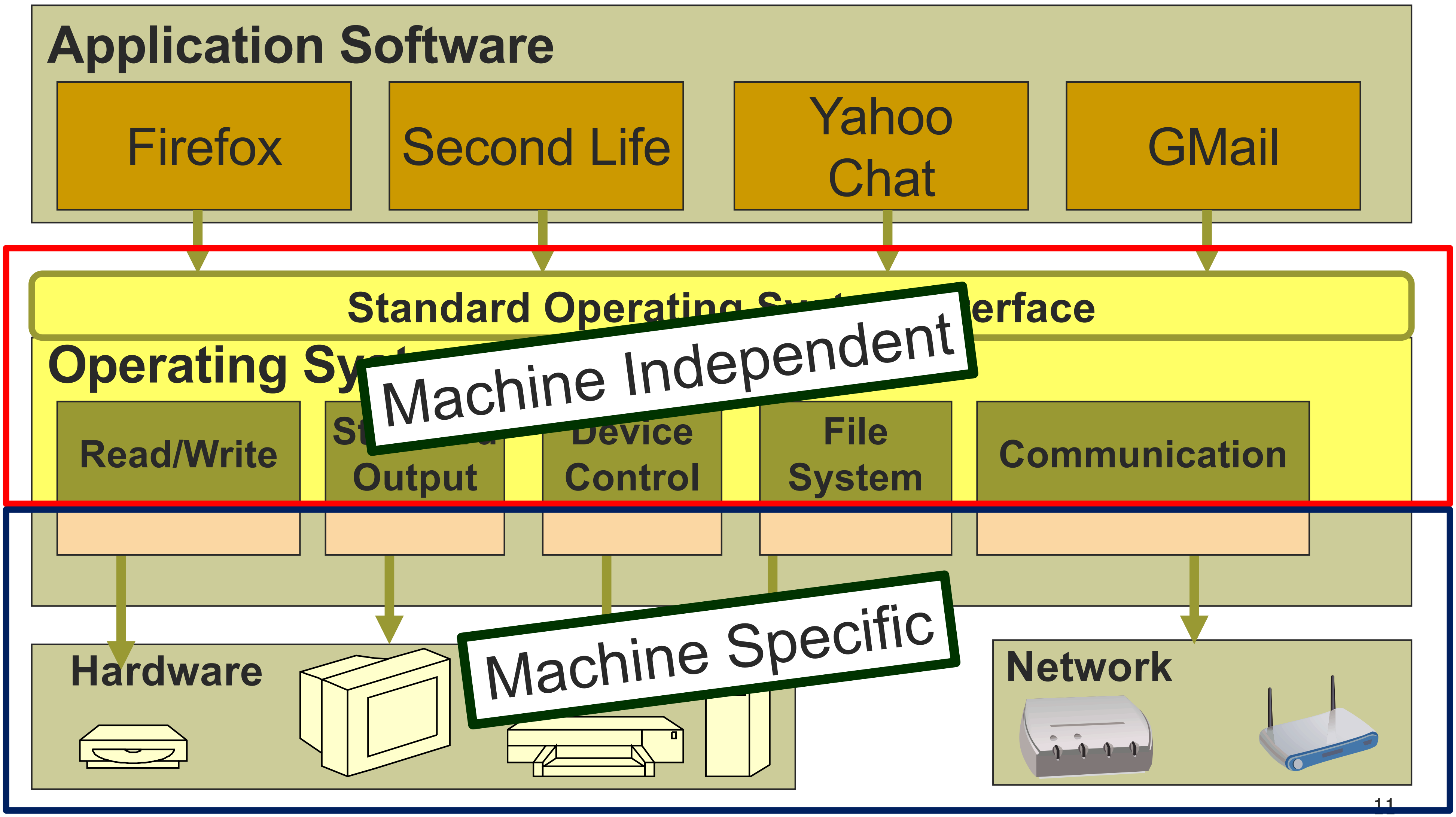
Delegate Common Functions



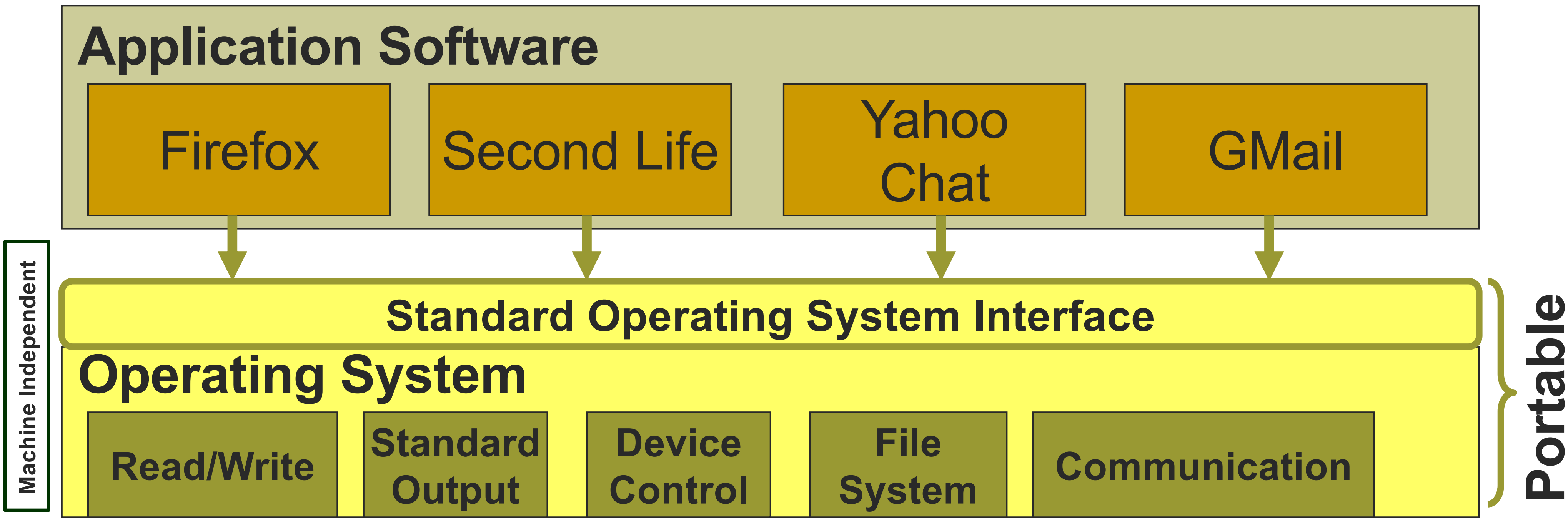
Export a Standard Interface



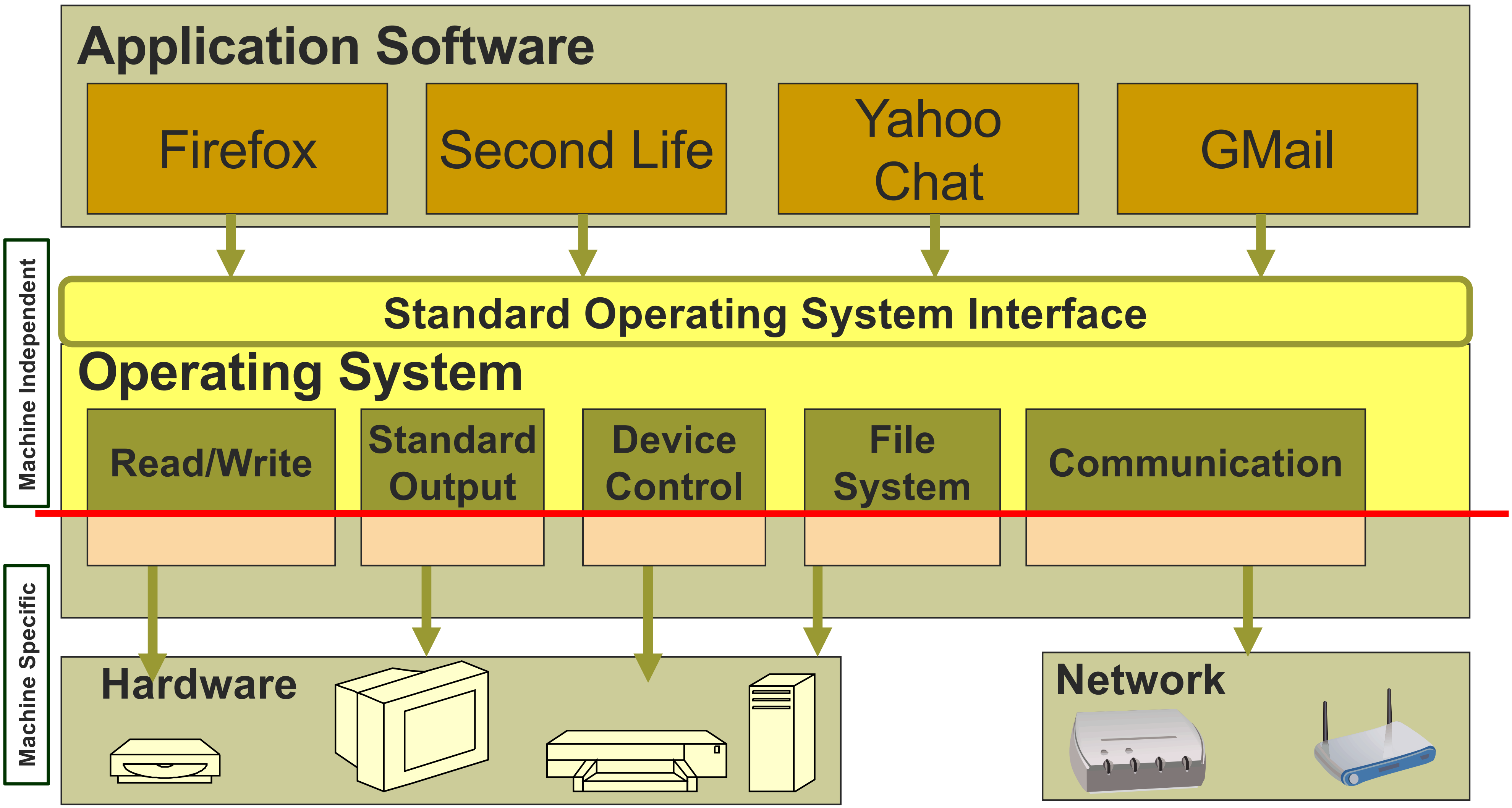
Goal: Increase Portability



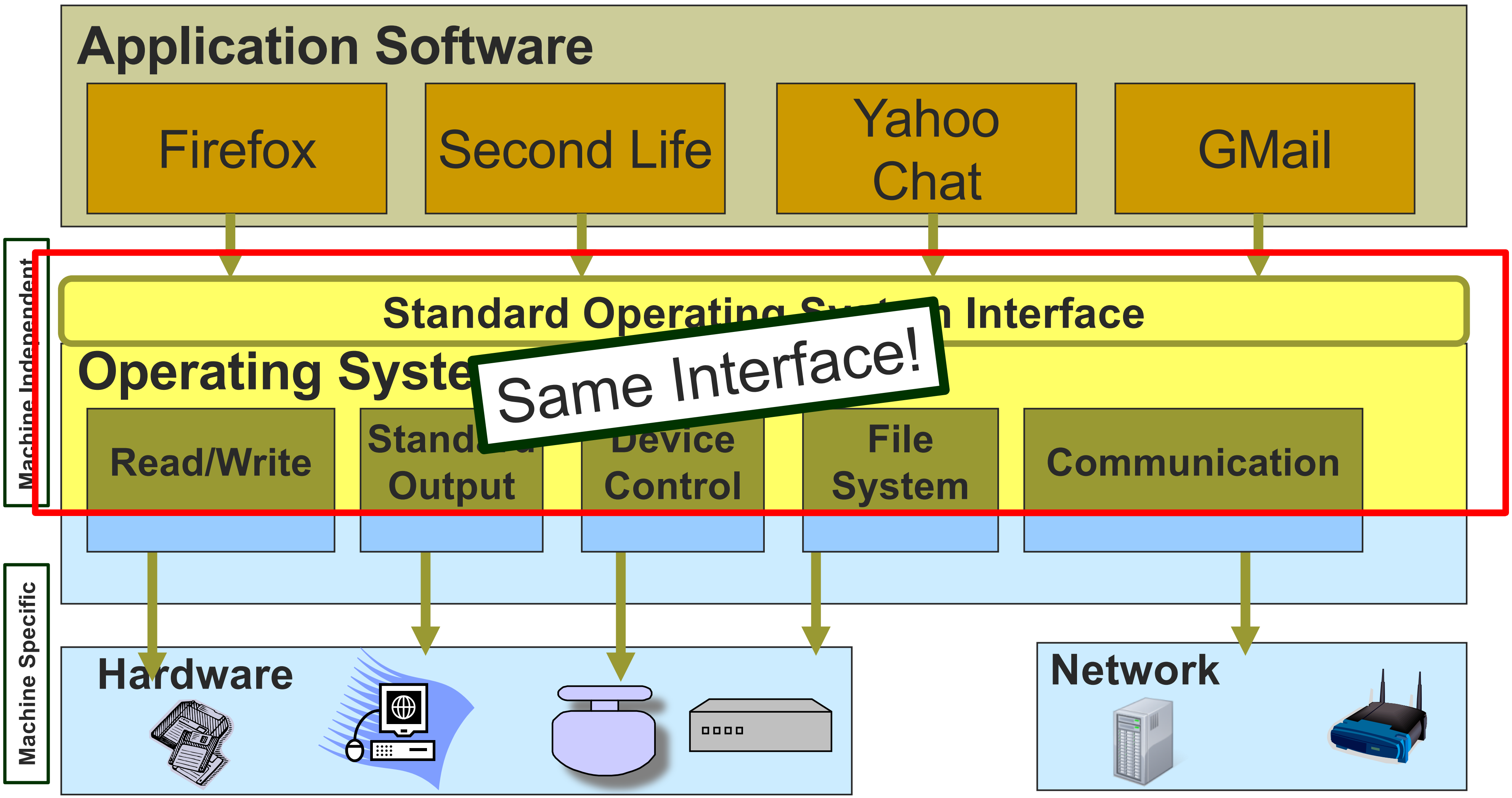
Machine Independent = Portable



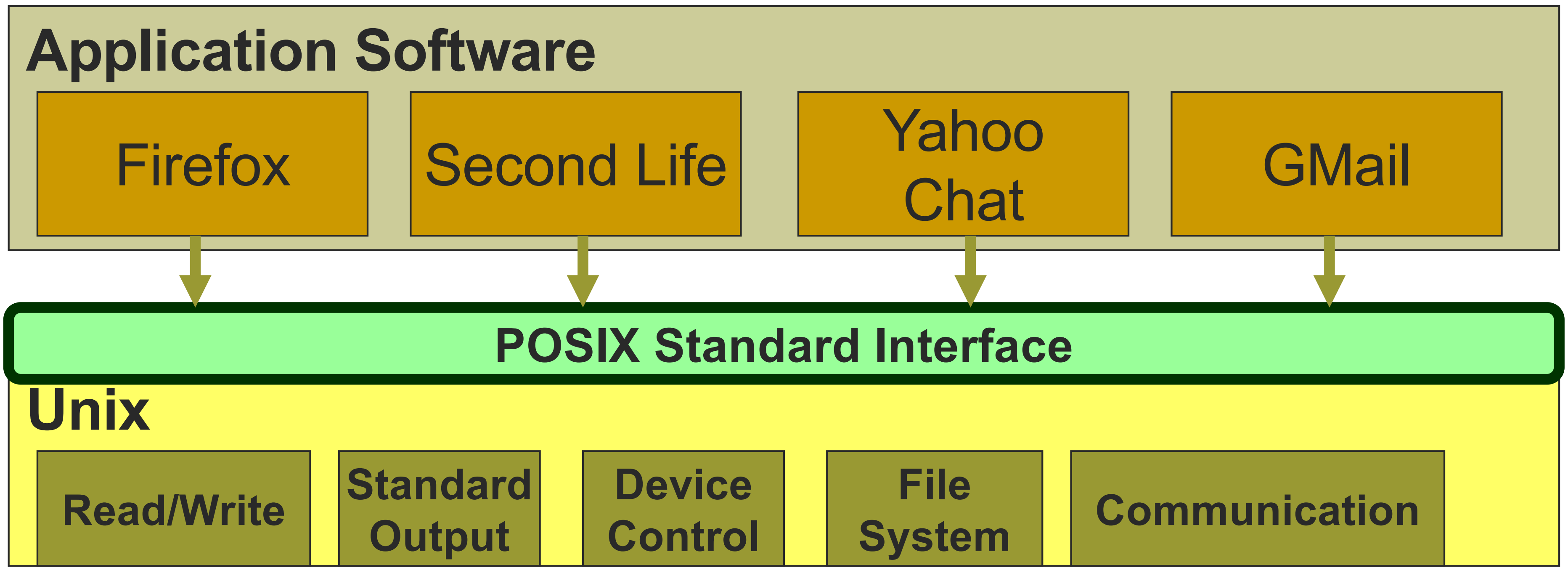
OS Runs on Multiple Platforms



OS Runs on Multiple Platforms



POSIX
The UNIX Interface Standard



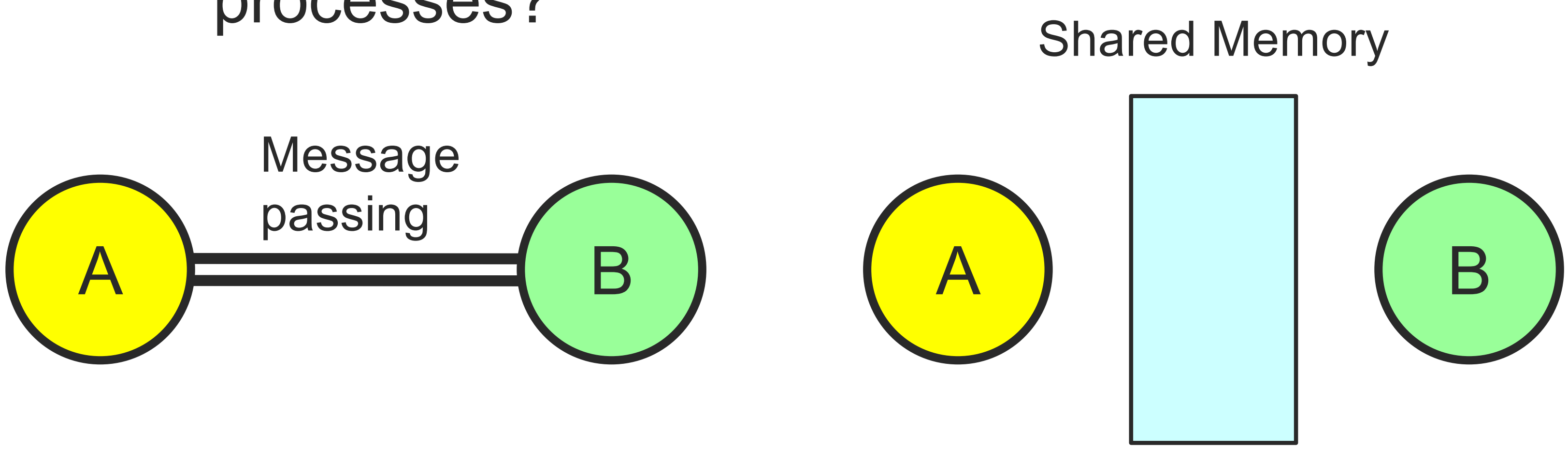
Operating System Concepts

- Process
  - An executable instance of a program
  - Only one process can use a (single-core) CPU at a time

- How is a program different from a process?
  - a program is a passive collection of instructions;
  - a process is the actual execution of those instructions; each process has a state to keep track of its execution
  - Several processes may be associated with the same program and share the same read-only code segment; for example, opening up several instances of the same program (like terminal) often means more than one process is being executed.

Operating System Concepts

- Inter-process Communication
  - Now process A needs to exchange information with process B
  - How would you enable communication between processes?



Posix.4 scheduling interfaces

- The real-time scheduling interface offered by POSIX.4 (available on Linux kernel)

- Each process can run with a particular scheduling policy and associated scheduling attributes. Both the policy and the attributes can be changed independently. POSIX.4 defines three policies:

- SCHED_FIFO: preemptive, priority-based scheduling.
- SCHED_RR: Preemptive, priority-based scheduling with quanta.
- SCHED_OTHER: an implementation-defined scheduler

Posix.4 scheduling interfaces

- SCHED_FIFO: preemptive, priority-based scheduling.
- The available priority range can be identified by calling:
  sched_get_priority_min(SCHED_FIFO) ➔ Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_FIFO); ➔ Linux 2.6 kernel: 99

- *SCHED_FIFO* can only be used with static priorities higher than 0, which means that when a *SCHED_FIFO* process becomes runnable, it will always preempt immediately any currently running normal *SCHED_OTHER* process. *SCHED_FIFO* is a simple scheduling algorithm without time slicing.

- A process calling **sched_yield** will be put at the end of its priority list. No other events will move a process scheduled under the *SCHED_FIFO* policy in the wait list of runnable processes with equal static priority. A *SCHED_FIFO* process runs until either it is blocked by an I/O request, it is preempted by a higher priority process, it calls **sched_yield,** or it finishes.

Posix.4 scheduling interfaces

- SCHED_RR: preemptive, priority-based scheduling with quanta.
- The available priority range can be identified by calling:
  sched_get_priority_min(SCHED_RR) ➔ Linux 2.6 kernel: 1
  sched_get_priority_max(SCHED_RR); ➔ Linux 2.6 kernel: 99

- *SCHED_RR* is a simple enhancement of *SCHED_FIFO*. Everything described above for *SCHED_FIFO* also applies to *SCHED_RR*, except that each process is only allowed to run for a maximum time quantum. If a *SCHED_RR* process has been running for a time period equal to or longer than the time quantum, it will be put at the end of the list for its priority.

- A *SCHED_RR* process that has been preempted by a higher priority process and subsequently resumes execution as a running process will complete the unexpired portion of its round robin time quantum. The length of the time quantum can be retrieved by **sched_rr_get_interval**.

Posix.4 scheduling interfaces

- SCHED_OTHER: an implementation-defined scheduler

- **Default Linux time-sharing scheduler**
- *SCHED_OTHER* can only be used at static priority 0. *SCHED_OTHER* is the standard Linux time-sharing scheduler that is intended for all processes that do not require special static priority real-time mechanisms. The process to run is chosen from the static priority 0 list based on a dynamic priority that is determined only inside this list.

- The dynamic priority is based on the nice level (set by the **nice** or **setpriority** system call) and increased for each time quantum the process is ready to run, but denied to run by the scheduler. This ensures fair progress among all *SCHED_OTHER* processes.

Posix.4 scheduling interfaces

- **<u>Do not forget!!!!</u>**

  ➔ a non-blocking end-less loop in a process scheduled under *SCHED_FIFO* or *SCHED_RR* will block all processes with lower priority forever, a software developer should always keep available on the console a shell scheduled under a higher static priority than the tested application. This will allow an emergency kill of tested real-time applications that do not block or terminate as expected.

- Since *SCHED_FIFO* and *SCHED_RR* processes can preempt other processes forever, only root processes are allowed to activate these policies under Linux.

Posix.4 scheduling interfaces

```
#include <sched.h>
#include <sys/types.h>
#include <stdio.h>

int             fifo_min, fifo_max;
int             sched, prio, i;
pid_t           pid;
struct sched_param  attr;

main()
{
        fifo_min = sched_get_priority_min(SCHED_FIFO); fifo_max = sched_get_priority_max(SCHED_FIFO);

        printf("\n Scheduling informations: input a PID?\n");
        scanf("%d", &pid);
        sched_getparam(pid, &attr);
        printf("process %d uses scheduler %d with priority %d \n", pid,
        sched_getscheduler(pid), attr.sched_priority);

        printf("\n Let's modify a process sched parameters: Input the PID, scheduler type, and priority \n");
        scanf("%d %d %d", &pid, &sched, &prio);

        attr.sched_priority = prio;
        i = sched_setscheduler(pid, sched, &attr);
}
```

POSIX.4 Real Time Clock & timers

- Linux provides an implementation of POSIX.4 real-time clock and timers. Timers (not to be confused with Timer/Counter units on the dsPic microcontroller) can be used to send a signal to a process after a specified period of time has elapsed.

- **Timers may be used in one of two modes: one-shot or periodic:**
  - when a **one-shot timer** is set up, a value time is specified. When that time has elapsed, the operating system sends the process a signal and deletes the timer.

  - when a **periodic timer** is set up, both a value and an interval time are specified. When the value time has elapsed, the operating system sends the process a signal and reschedules the timer for interval time in the future. When the interval time has elapsed, the OS sends another signal and again reschedules the timer for interval time in the future. This will continue until the process manually deletes the timer.

- By default a timer will send the SIGALRM signal. If multiple timers are used in one process, however, there is no way to determine which timer sent a particular SIGALRM. Therefore, an alternate signal, such as SIGUSR1, may be specified when the timer is created.

POSIX.4 Real Time Clock & timers

- **Resolution of timers:**

- Timers are maintained by the operating system, and they are (usually) only checked periodically. A timer that expires between checks will be signaled (and rescheduled if periodic) at the next check. As a result, a process may not receive signals at the exact time(s) that it requested.

- The period at which the timers are checked, called the **clock resolution**, is operating system and hardware dependent (~1 msec in PC without add-on high resolution real time clock cards). The actual value can be determined at runtime by calling clock_getres() on the system-wide real-time clock (CLOCK_REALTIME). According to POSIX.4, there is at least 1 real-time clock (CLOCK_REALTIME)

POSIX.4 Real Time Clock & timers

- **High-Resolution Timers in Linux (see man 7 time):**

- Before Linux 2.6.21, the accuracy of timer and sleep system calls was limited by the size of the jiffy (resolution of the software clock maintained by the kernel).

- Since Linux 2.6.21, Linux supports high-resolution timers (HRTs), optionally configurable via CONFIG_HIGH_RES_TIMERS. On a system that supports HRTs, the accuracy of sleep and timer system calls is no longer constrained by the jiffy, but instead can be as accurate as the hardware allows (even nanosecond accuracy is typical of modern hardware). You can determine whether high-resolution timers are supported by checking the resolution returned by a call to clock_getres(2) or looking at the "resolution" entries in /proc/timer_list.

POSIX.4 Real Time Clock & timers

- **Operations:**

- Create_timer() is used to create a new timer. As with clock_getres(), the system-wide real-time clock (CLOCK_REALTIME) should be used. The following code shows how to create a timer that sends the default SIGALRM signal.

```
timer_t timer1;

// Create a new timer that will send the default SIGALRM signal.
if (timer_create(CLOCK_REALTIME, NULL, &timer1) != 0)
{
        // If there is an error, print out a message and exit.
        perror("timer_create");
        exit(1);
}
```

NULL specifies that default SIGALARM will be delivered!

POSIX.4 Real Time Clock & timers

- **Operations:**
- The timer_settime() function is used to schedule a timer. The struct itimerspec definition taken from /usr/include/linux/time.h is seen here.
- The **it_value** member sets the time until the timer first expires. If it is set to 0, the timer will never go off. The **it_interval** member sets the period of the timer after it first expires. If it is set to 0, the timer will be one-shot.

```
struct itimerspec {
        struct timespec it_interval;    /* timer period */
        struct timespec it_value;       /* timer expiration */
};

struct timespec {
        time_t tv_sec;          /* seconds */
        long   tv_nsec;         /* nanoseconds */
};
```

POSIX.4 Real Time Clock & timers

- **Operations:**
- Following is an example of scheduling timer1 (created in a preceding example) to go off in 2.5 seconds, and then every 100 milliseconds thereafter.

```
struct itimerspec timer1_time;

// The it_value member sets the time until the timer first goes off (2.5 seconds).
// The it_interval member sets the period of the timer after it first goes off (100 ms).
timer1_time.it_value.tv_sec     = 2;          // 2 seconds
timer1_time.it_value.tv_nsec    = 500000000;  // 0.5 seconds (5e8 nanoseconds)
timer1_time.it_interval.tv_sec  = 0;          // 0 seconds
timer1_time.it_interval.tv_nsec = 100000000;  // 100 milliseconds (1e8 nanoseconds)

// Schedule the timer.
if (timer_settime(timer1, 0, &timer1_time, NULL) != 0)
{
        // If there is an error, print out a message and exit.
        perror("timer_settime");
        exit(1);
}
```

Pointer to old timer spec!

It specifies a “relative timer”: it does not use absolute time!

POSIX Real Time Clocks

- POSIX.4 defines the structure of time representation. There is at least 1 real time clock. We can check the current time with **clock_gettime** and check the clock resolution with **clock_getres**. See the following example:

```
#include <stdio.h>
#include <time.h>

main() {

  struct timespec  current_time, clock_resolution;
  clock_gettime(CLOCK_REALTIME, &current_time);

  printf("current time in CLOCK_REALTIME is %ld sec, %ld nsec \n",
        current_time.tv_sec,                    // the second portion
        current_time.tv_nsec);                  // the fractional portion in nsec

  clock_getres(CLOCK_REALTIME, &clock_resolution);

    printf("CLOCK_REALTIME's resolution is %ld sec, %ld nsec \n", clock_resolution.tv_sec, clock_resolution.tv_nsec);
}
```

```
mcaccamo@versilia:~/cs431_f16_teaching/code/lecture7$ ./timer

current time in CLOCK_REALTIME is 1473865424 sec, 743168251 nsec
CLOCK_REALTIME's resolution is 0 sec, 1 nsec

mcaccamo@versilia:~/cs431_f16_teaching/code/lecture7$
```