What is a Process?

Answer 1:

- A process is an abstraction of a program in execution.

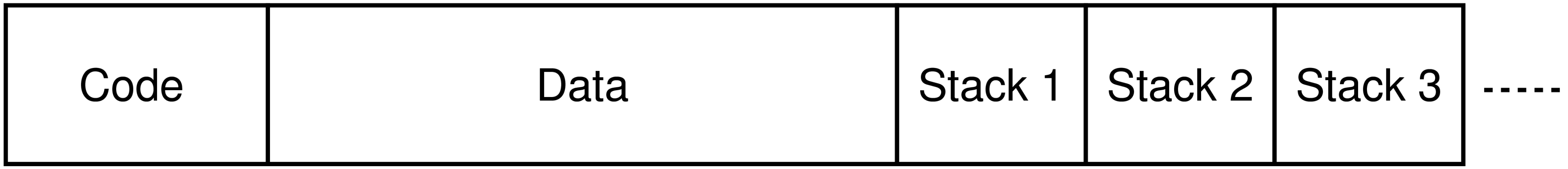
Answer 2:

- an address space
- one or more threads of execution
- resources associated with the running program, such as:
  - open files
  - sockets
  - locks held by the program

A process with one thread is a *sequential* process. A process with more than one thread is a *concurrent* process.

What is an Address Space?

A process' address space is its view of the computer's primary memory (RAM). It contains the process' code, its data, and stack space for each of its threads.



It can also contain other data, such as the content of entire files, via memory mapping (mapping files into an address space: `mmap`).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

What is a Thread?

A thread represents the control state of an executing program.

Each thread has a *context*:

- the contents of the processor's registers, including program counter and stack pointer
- other state information, such as priority level and execution privileges
- a stack (located in the process' address space)

Multiple threads within the same process can be thought of as processes that share their address space (code, data, open files, etc.).

Whether there is an implementation difference between a thread and a process depends on the OS. Nachos: Yes. Linux: Usually no.

Execution Privileges and the OS Kernel

In most systems, there are two types of processes:

- ordinary user processes without special privileges;
- kernel processes with full access to the whole system.

The operating system kernel is a program. It has code and data like every other program.

A process can switch from user mode to kernel mode by performing a system call (syscall), executing code in the kernel.

When a user process performs a syscall, security checks have to be performed to make sure the process has permission to do what it is trying to do.

Some processes always run in kernel mode: kernel device drivers, I/O daemons, interrupt handlers.

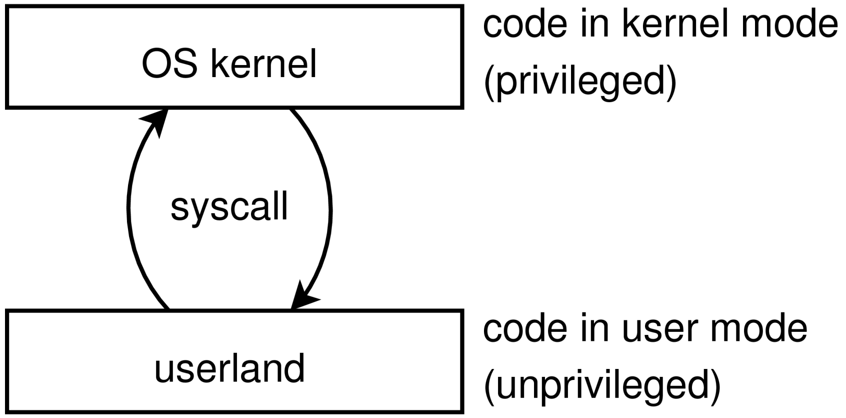
Execution Privileges and the OS Kernel

The kernel has privileged access to hardware and protects and isolates itself from user processes.

The exact set of privileges is different from platform to platform. Some examples:

- execute special instructions (e.g., halt)
- manipulate the processor state (e.g., 32-bit vs. 64-bit)
- ability to access the entire main memory of the computer

No user process can execute kernel code or read/write kernel data, except through controlled mechanisms (syscalls).


OS kernel
code in kernel mode (privileged)
syscall
userland
code in user mode (unprivileged)

System Calls

System calls are the interface between userland and OS kernel.

A process uses system calls to request OS services.

A process can use a system call to:

- get the current time (`gettimeofday`);
- open/close a file (`open`/`close`);
- send a message over a pipe (`send`);
- create another process (`fork`/`clone`);
- change its own scheduling priority (`setpriority`);
- change the size of its address space;
- change the user it is associated with (`setuid`);
- terminate itself (`exit`).

System Calls

System calls are usually realized through a special instruction, e.g., the MIPS `syscall` instruction.

What happens on a system call?

- The processor is switched to privileged execution mode.
- Key parts of the current thread context (program counter, stack pointer, etc.) are saved.
- The thread's context is changed:
  - The PC is set to a fixed memory address within the kernel's code.
  - The stack pointer is set to an address pointing to a stack in kernel space.

System Calls

Execution and Return

After issuing the system call, the calling thread executes a system call handler, which is part of the kernel, in kernel mode.

The system call handler determines which service the calling thread wants and whether it has sufficient permissions.

When the kernel is finished, it restores the key parts of the calling thread's context and switches the processor back from kernel mode into unprivileged user mode.

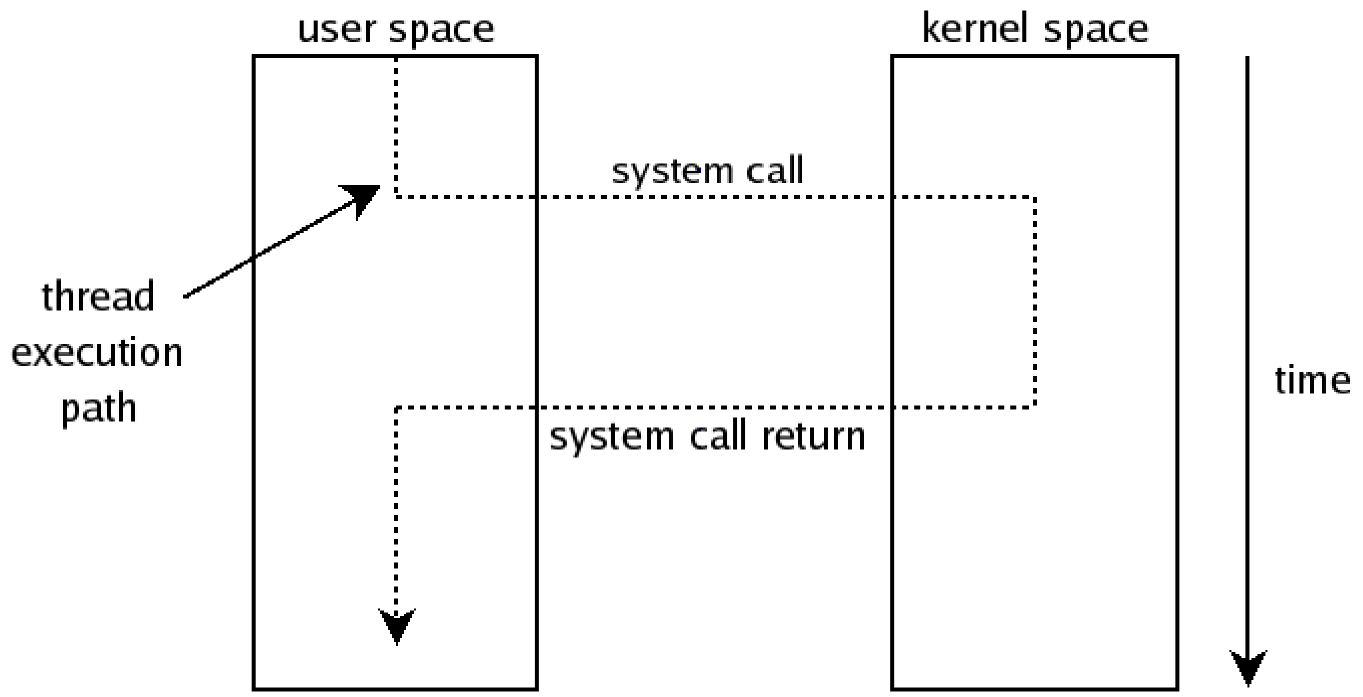
The thread continues executing code in the calling program (in user space).

System Calls in Nachos

In NachOS, system calls are handled by the `ExceptionHandler` function in `code/userprog/exception.cc`:

```
void ExceptionHandler(ExceptionType which) {
  int type, vaddr, returnval = 0;
  int type = kernel->machine->ReadRegister(2);
  switch (which) {
    case SyscallException:
      switch (type) {
        case SC_Halt:
          kernel->interrupt->Halt();
        case SC_Exit:
          status = kernel->machine->ReadRegister(4);
          Terminator(status);
          break;
        case SC_Create:
          vaddr = kernel->machine->ReadRegister(4);
          returnval = CreateHandler(vaddr);
          break;
      }
  }
}
```

System Call Diagram



The syscall handler is executed by the same thread that issued the system call. However, it has additional privileges when running in kernel mode.

Exceptions

In addition to system calls, exceptions are another way of transferring control from a user space process to the kernel.

Exceptions occur when a thread is trying to execute an instruction that is either not permitted or requires some additional work before it can be executed:

- arithmetic error (e.g., division by zero);
- illegal instruction (garbage bit sequence);
- memory protection violation (aka segmentation fault);
- page fault (page needs to be loaded into RAM first).

Exceptions are detected by the hardware and lead to the execution of a kernel exception handler by the thread causing the exception.

Exceptions

When an exception occurs, control is transferred to a fixed point in the kernel.

Transfer of control is realized just like in the case of system calls.

When an exception occurs, the kernel has to decide how to deal with it. It may decide to kill the thread (or process) responsible for the exception, it may decide to just ignore the instruction that caused the exception, or anything else.

In Nachos, there is a single, unified handler function for both exceptions and system calls (cf. slide “System Calls in Nachos”).

Interrupts

Interrupts are a third way of transferring control to the kernel.

Interrupts are similar to exceptions, but caused by hardware instead of the execution of a program. Examples:

- timer interrupt (periodic, e.g. every 10 ms);
- disk controller informs kernel that it has finished writing some data to disk;
- network controller reports the arrival of a network packet;
- keyboard controller notifies kernel that a key has been pressed;
- ...

Interrupt handling is very similar to exception handling. Slight difference: The interrupt handler usually does not have a thread context. In some systems (e.g., Linux), this has implications on the actions an interrupt handler may perform.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Implementation of Processes

The kernel maintains a special data structure, the *process table*, that contains information about all processes in the system.

Information about individual processes is stored in a structure sometimes called *process control block* (PCB).

Per-process information in a PCB may include:

- process identifier (PID) and owner (UID)
- current process state (runnable, blocked) and other information related to process scheduling
- lists of available resources – open files, locks, network sockets, ...
- accounting information (CPU time etc.)

In NachOS, if you do not want to introduce a new data structure, you can use the `AddrSpace` class as PCB.

Process Creation

The details of how to create a new process in order to run another program differ from system to system.

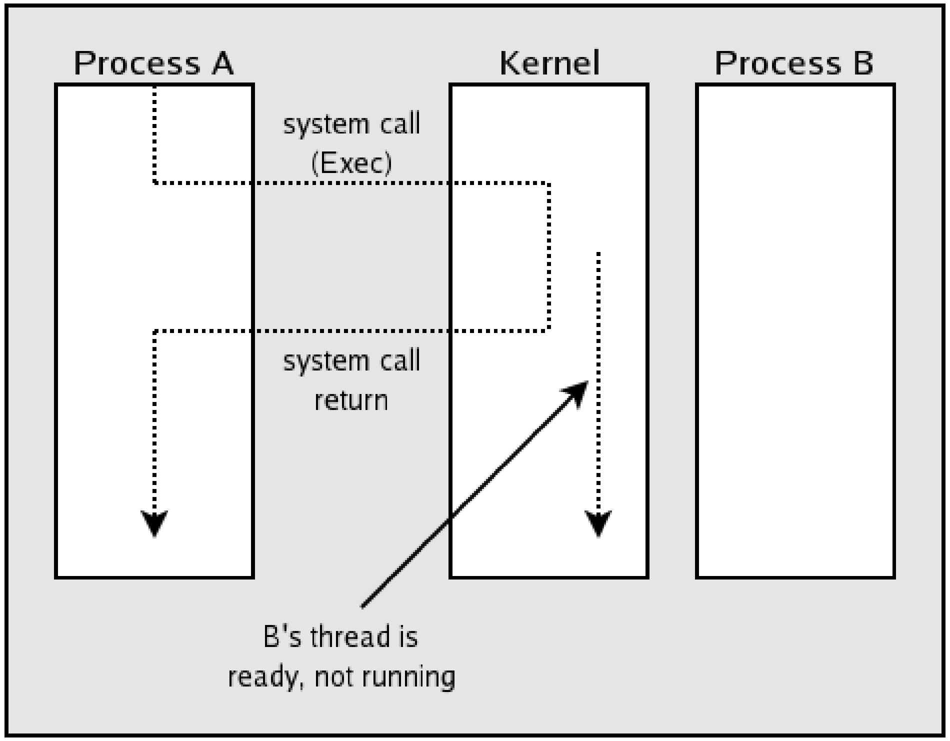
Method 1: Execute

- Issue a system call. The operating system creates a new process and starts the given application (binary executable) in the new process.

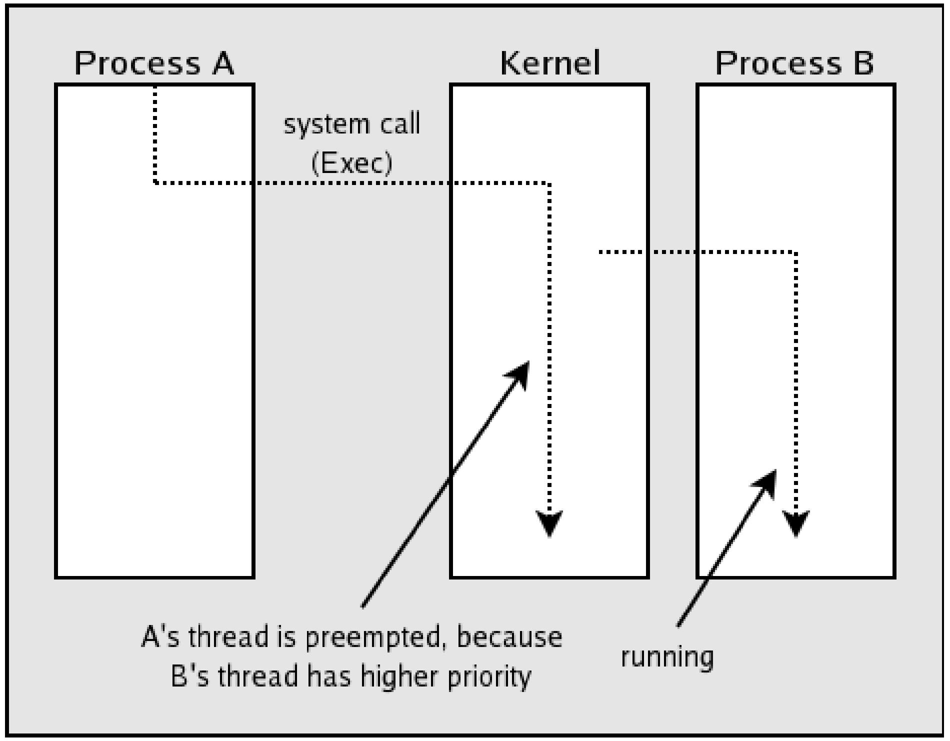
Method 2: Fork & Execute

- First system call: Fork. The operating system creates a new process that is an exact copy of the old process (however, with different process ID).
- Second system call (in new process): Execute. The operating system replaces the program in the calling process by the new program.

Process Creation in Nachos



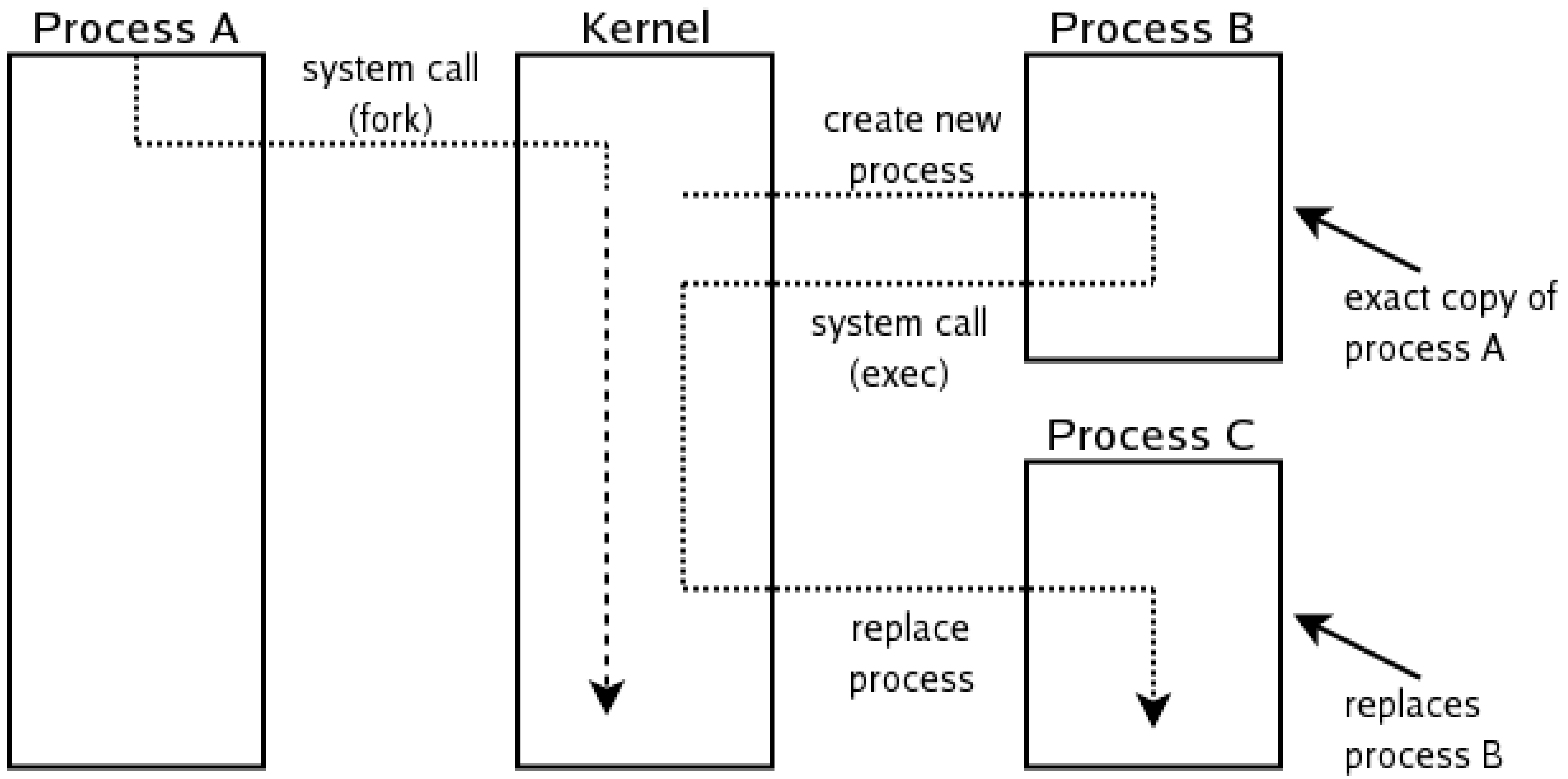
Scenario I: getPriority(A) >= P_NORMAL


Process A
Kernel
Process B
system call (Exec)
A's thread is preempted, because B's thread has higher priority
running

Scenario II: getPriority(A) < P_NORMAL

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Process Creation in Real Operating Systems



Multiprogramming

Multiprogramming:
Allowing multiple processes to exist at the same time.

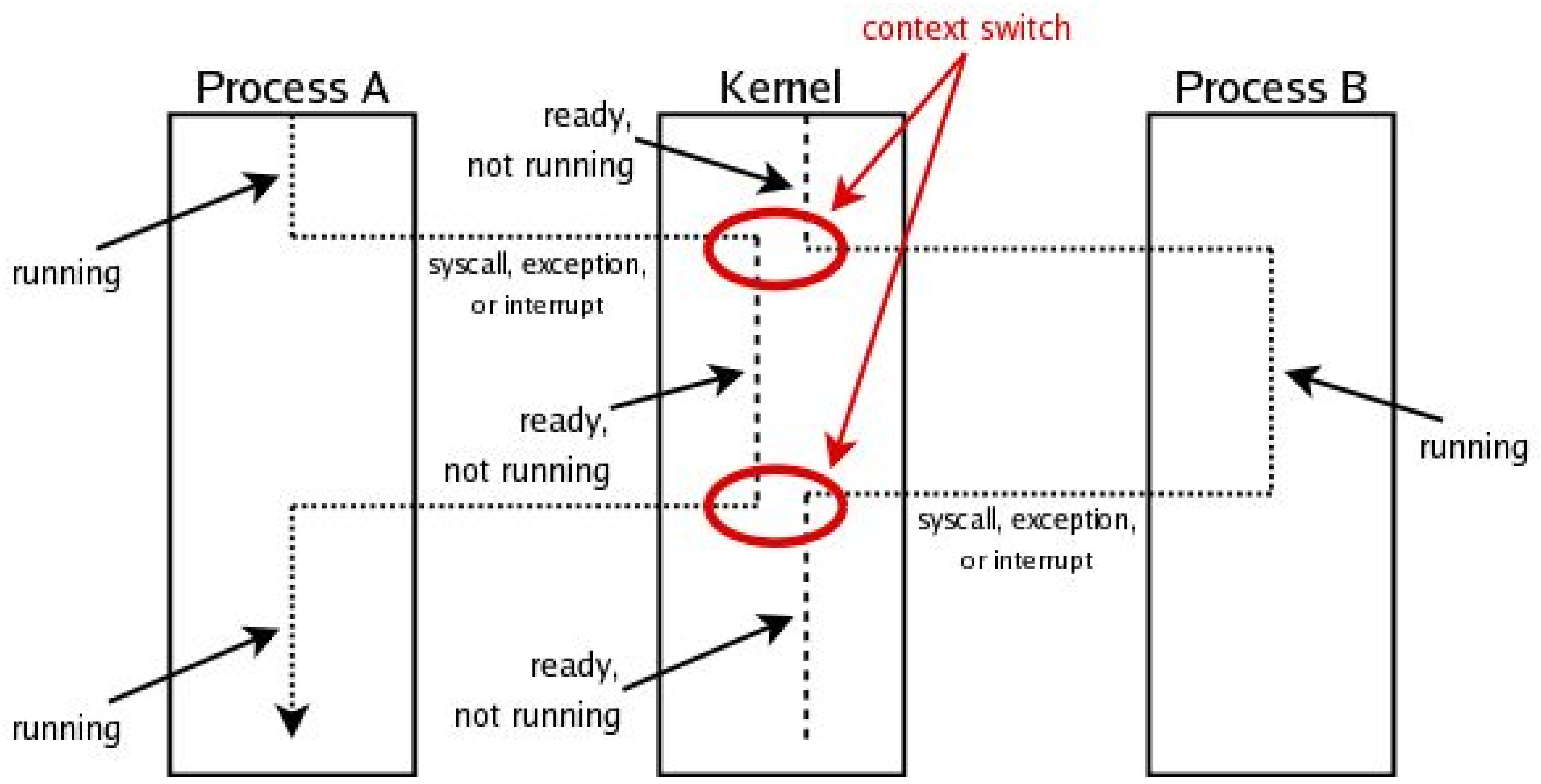
Most modern operating systems support multiprogramming (also referred to as multitasking).

Processes need to share:

- available main memory;
- hardware devices: hard disk, keyboard, etc.;
- CPU time.

The operating system ensures the processes are isolated from each other.

CPU Timesharing



Stefan Buettcher
<sbuettch@uwaterloo.ca>

Processor Scheduling

Only one thread can run on a processor at a given time (exceptions: dual-code, hyper-threading).

Transferring the control over the CPU between two threads is called a *context switch*.

How is a context switch realized? How is it triggered?

A context switch can take place whenever the kernel has control over the CPU.

Windows 3.x:
Cooperative multitasking – The kernel can perform a context switch whenever a system call takes place.

Windows 95 and later (and Linux since version 1.0):
Preemptive multitasking – The kernel can perform a context switch after the system call, an exception, or an interrupt (timer!).

Preemptive Scheduling

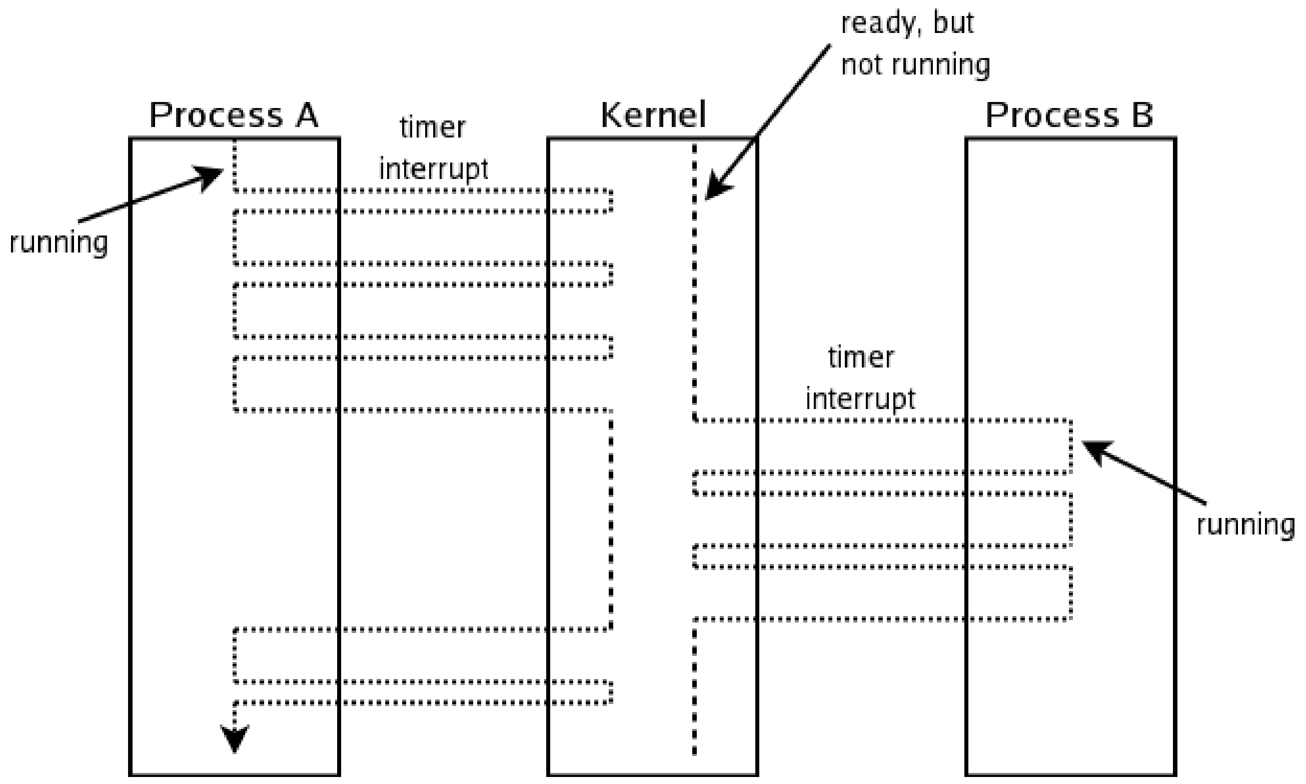
Simple preemptive scheduling policy: Round-robin with time slices.

- The kernel maintains a list of *ready* (aka *runnable*) threads.
- The first thread in the list is allowed to run.
- When the running has run for a certain amount of time (called the *scheduling quantum*, or *time slice*), it is preempted.
- The preempted thread goes to the back of the *ready* list; the thread at the front of the *ready* list is allowed to run.

How do we know when a thread's time slice has expired, and how do we transfer control to the kernel?

- Use timer to periodically check how much time has expired.
- Every timer interrupt transfers control of the CPU to the kernel; the kernel can decide whether it is time to preempt the current thread and transfer control to another thread.

Preemptive Scheduling



Frequency of the timer interrupt differs from system to system.
Linux (before 2.6): 100 Hz. Since 2.6.x: 1000 Hz or 250 Hz.

Default time slices differ from system to system.
Linux (since 2.6.x): Default time slice is 150 ms. However, scheduling in Linux is complicated and takes interaction with hardware devices into account. Interactive processes sometimes get higher priority.

Blocked Threads

The simple preemptive scheduling strategy maintains a list of *ready* threads, but what does it mean for a thread to be ready?

Definition – A thread is ready when it is not blocked.

A thread can be blocked because it is

- waiting for data from the hard disk;
- waiting for keyboard input;
- waiting for some resource to become available;
- waiting for another thread to leave a critical section.

The scheduler will only allocate the processor to threads that are not blocked.

Implementing Blocking

A thread usually becomes blocked during the execution of a system call, e.g., requesting access to a hardware device.

Are there other ways for a thread to become blocked?

When the kernel recognizes that a thread *T* faces a delay, it can block that thread:

- mark *T* as blocked, do not put it into the ready list
- choose a ready thread to run
- when *T* is not blocked any more (awaited event occurred), put it back into the ready queue, potentially switch to it right away

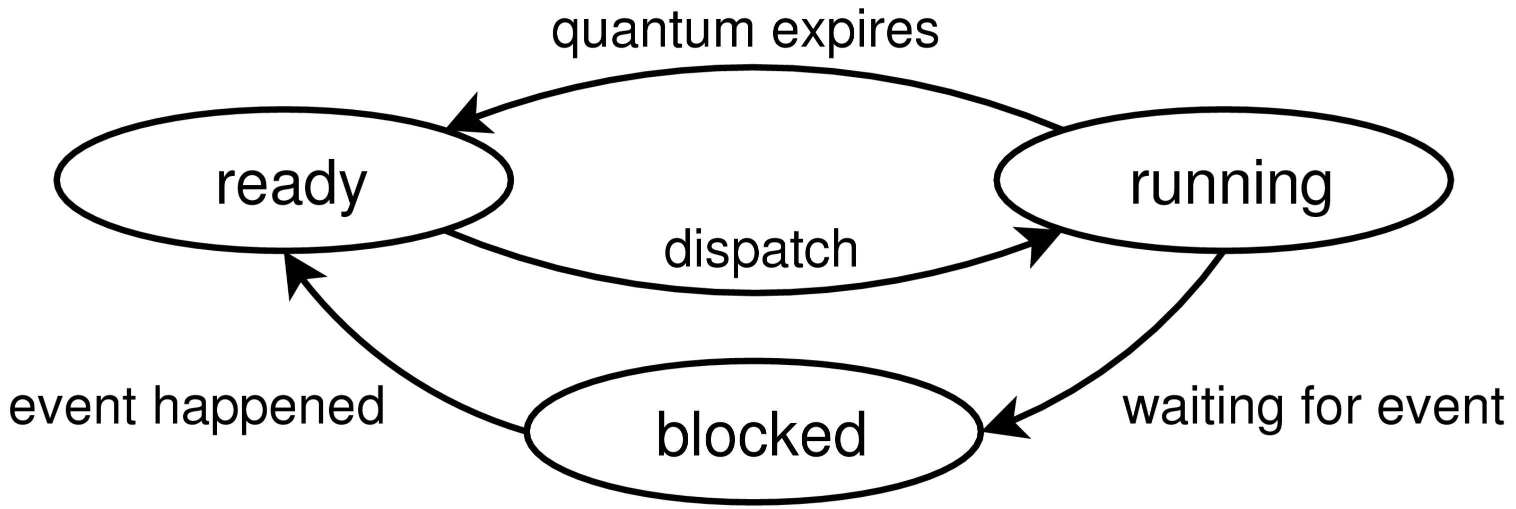
The kernel does not have to block a thread. Examples: Disk I/O scheduling; super-low-latency network applications.

Thread States

A summary of possible thread states:

- **Running** – currently executing (thread has the CPU)
- **Ready** – ready to execute, not waiting for anything
- **Blocked** – waiting for an event, not ready to execute

Thread state transition diagram:



Implementing Threads

Threads can be implemented either in the user program (library) or in the kernel. So far, we have assumed they are implemented in the kernel.

Three different ways of realizing threads:

- N:1 – multiple user-level threads (usually all threads in the same process) share one kernel thread.
- N:N – each user-level thread has its own kernel thread.
- N:M – some mixture.

The N:N variant is usually subsumed under N:M.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

User-Level Threads – N:1

All threads in the same process are represented by a single kernel thread.

They are called *user-level threads* because the kernel is unaware of their existence.

A user-level thread library provides functions for thread creation, termination, yielding, and synchronization.

Advantages:

Does not depend on a multi-threaded kernel.

Disadvantages:

Unfair scheduling. If a single thread is blocked, then the whole process is blocked.

Example: Java (early versions).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Kernel Threads – N:M

Each user-level thread has its own kernel thread (sometimes: a few user-level threads share the same kernel thread).

Advantages:

- The kernel is aware of the existence of all threads and can take it into account when making a scheduling decision.
- Switching between threads in the same process can be preemptive.
- When a thread is blocked, the other threads in the same process can still continue working.
- A process can use multiple CPUs (one per active thread).

Disadvantages:

Super-low-latency applications (again...): Context switch may cause some delay.

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Inter-Process Relationships

Processes are in certain relationships with each other.

The most important relationship: parent ⇔ child

- If process A creates process B (fork or execute), then A is referred to as B's parent. B is A's child.
- A process may have multiple children, but at most one parent.
- The set of all processes forms a tree (more accurately: a forest, as parent-child relationships can sometimes be terminated).

Other relationships:

- Process groups (e.g., broadcast signals to multiple processes).
- Owned-by-same-user (all processes owned by the same user).

Stefan Buettcher
<sbuettch@uwaterloo.ca>

Parent-Child Relationship

The parent process knows the process IDs of all its children (return value of `Exec` in Nachos; return value of `fork` in Linux/POSIX).

The parent process can wait for the termination of its child (or children).

- in Nachos: `Join(ProcessID)`
- in Linux: `wait(int*)` or `waitpid(pid_t, int*, int)`

In some systems, the parent is even expected to wait for the termination of its children and to check their exit codes:

> *On Unix operating systems, a zombie process or defunct process is a process that has completed execution but still has an entry in the process table, allowing the process that started it to read its exit status. In the term's colorful metaphor, the child process has died but has not yet been reaped.*

(http://en.wikipedia.org/wiki/Zombie_process)

Process Interface: Summmary

Process interface refers to the set of functions (system calls) that can be used to manipulate processes in the system.

Creation – create a new process (`Exec` in Nachos, fork & exec in other operating systems).

Destruction – terminate the current process (`Exit` in Nachos) and return an exit code ("0" means success; "!=0" means error).

Synchronization – wait for an event, e.g., the termination of some other process (`Join` in Nachos).

Replacement – replace an existing process by a new one (`exec` in Linux/POSIX).

Attribute Management – read or change process attributes, such as scheduling priority, address space, process ID, parent's process ID, owner's user ID, etc.

Demo

Real-life demo:

- Parent-child relationship.
- Zombie processes.
- Scheduling new process.
- The fork of death.

Stefan Buettcher
<sbuettch@uwaterloo.ca>