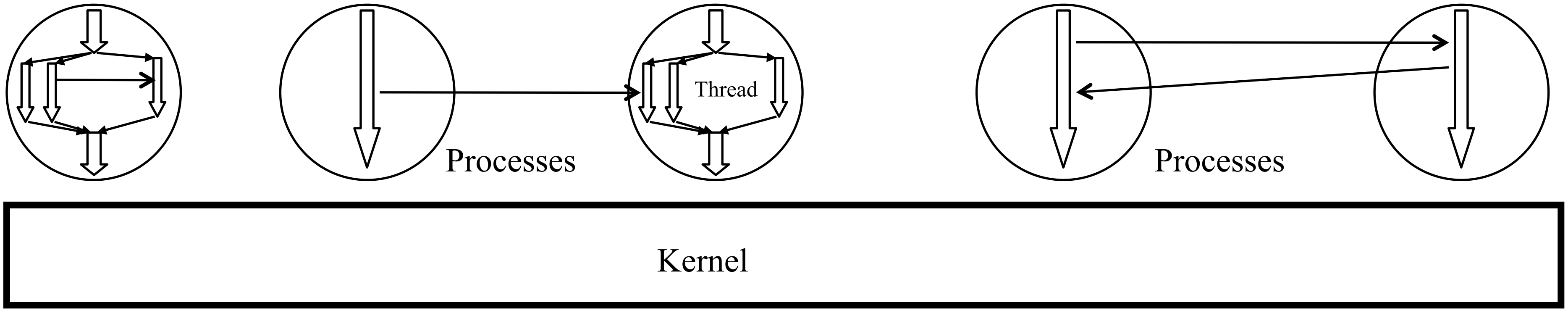
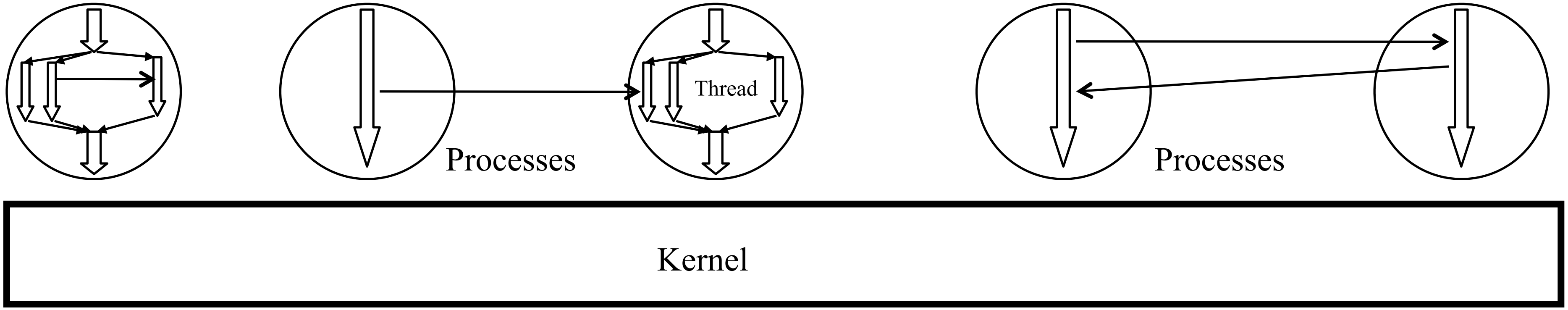
Processes and Non-Preemptive Scheduling

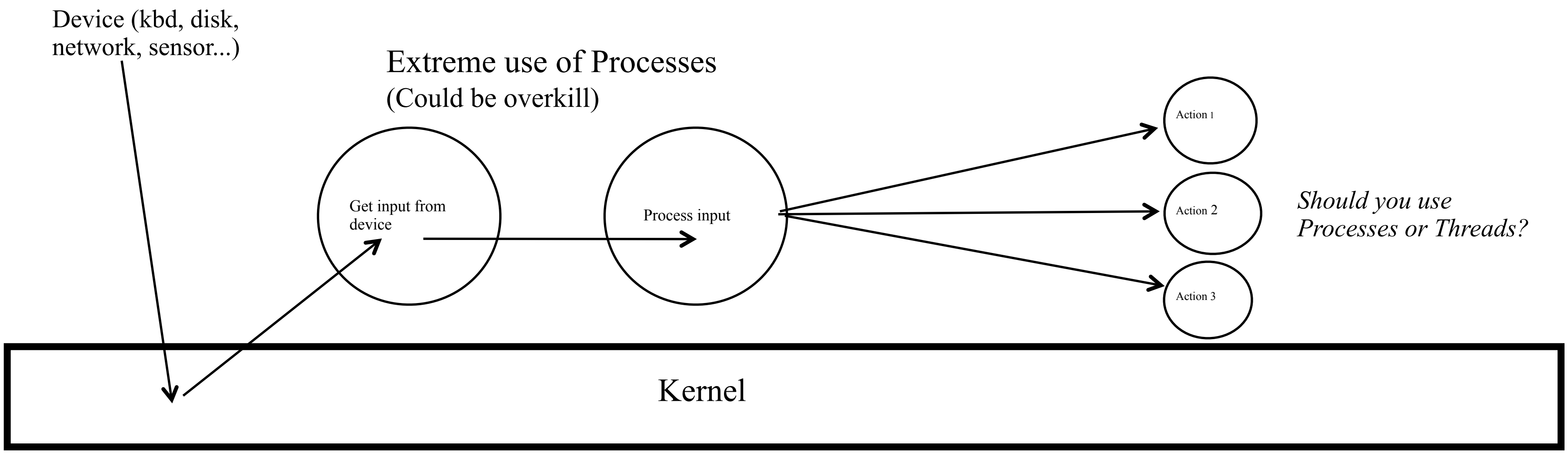
Otto J. Anshus

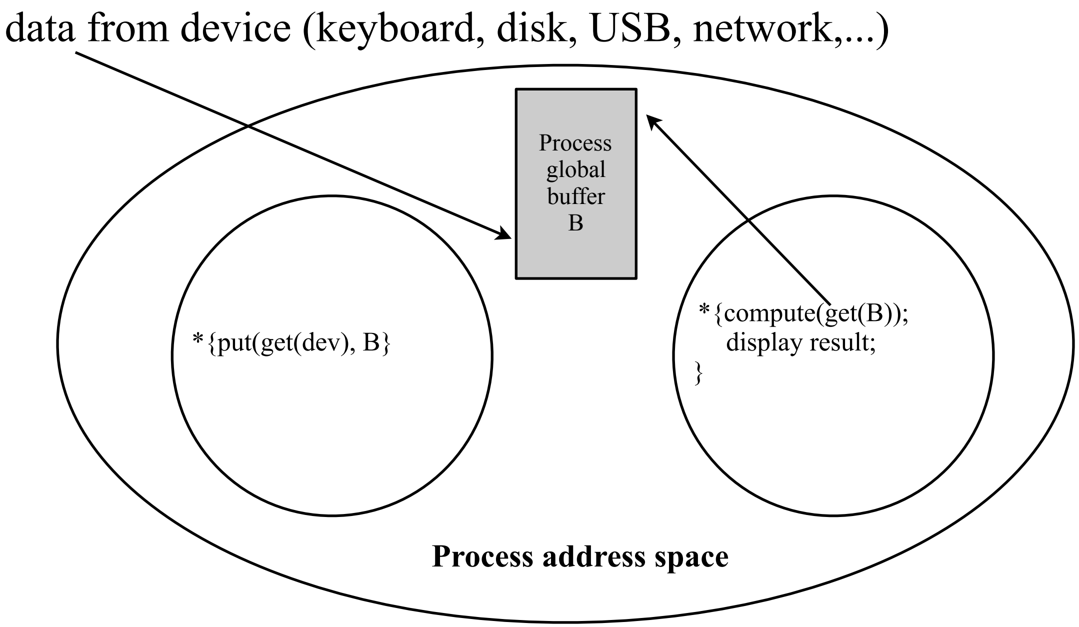
Concurrency and Process

- Challenge: Physical reality is Concurrent
  - "Concurrent software" may more simply than sequential be able to reflect this
  - We want to have many apps running on a single computer "at the same time"
    - Must share CPU, memory, I/O devices
    - Lots of interrupts/traps/exceptions/faults will happen
  - Options
    - let each app see the others and deal with it (each must fight or cooperate with the others, like cars in a city)
    - let each app believe it has the computer all alone (analogy: each car is all alone in the city (*however, the speed of the car can change at any time independently of the driver, including suddenly stopping/starting and crashing*)
- Trad. solution: Make the OS understand "process" and "threads" and give support to the processes and threads
  - Now we can decompose complex problems into simpler ones
    - Applications/computations are comprised of one or several processes
      - Cooperating processes need synchronization and communication (using **message passing**)
        - Each process comprised of one or several
          - Cooperating threads
          - Synchronization and communication (using **locks, semaphores, monitors**)
    - Deal with one smaller problem at a time: use a process or a thread for each
      - Drawback: performance?
      - Alternative: Event oriented model
    - Each process can now believe it has a computer to itself: it can be written as if this is indeed the case

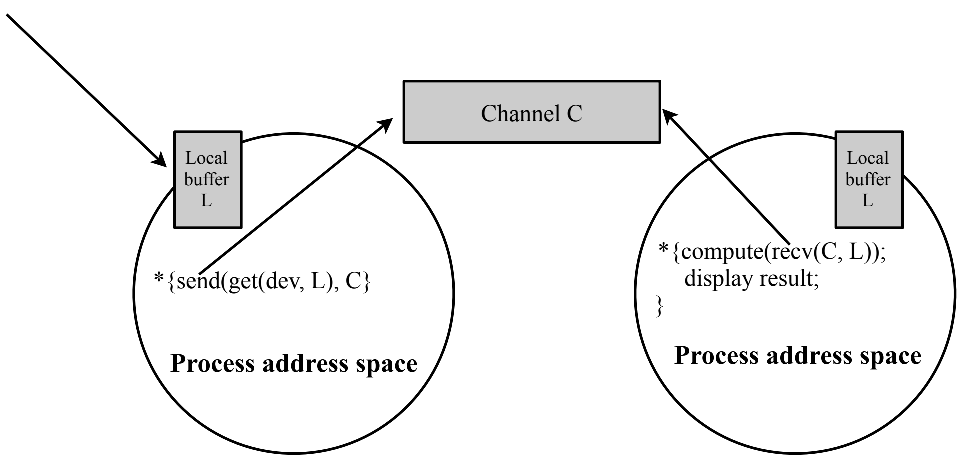

Thread
Processes
Processes
Kernel


Thread
Processes
Processes
Kernel

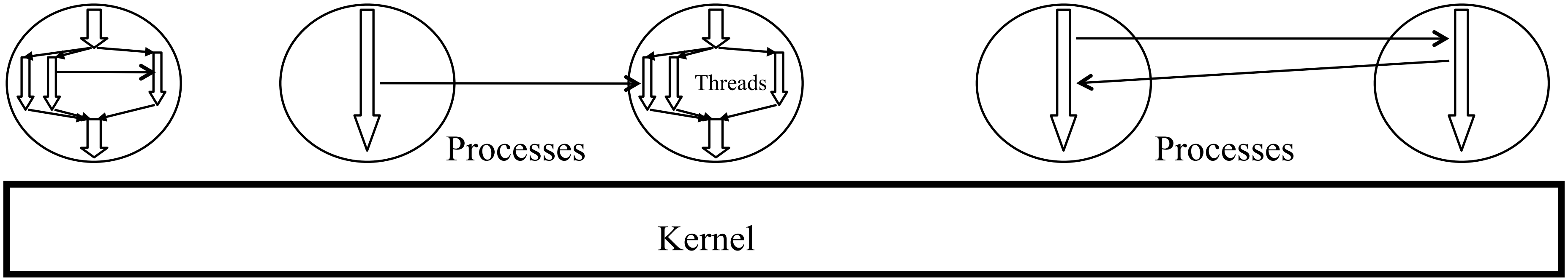

Device (kbd, disk, network, sensor...)
Extreme use of Processes
(Could be overkill)
Get input from device
Process input
Action 1
Action 2
Action 3
Should you use Processes or Threads?
Kernel



An application comprised of multiple threads inside a single process



An application comprised of multiple processes

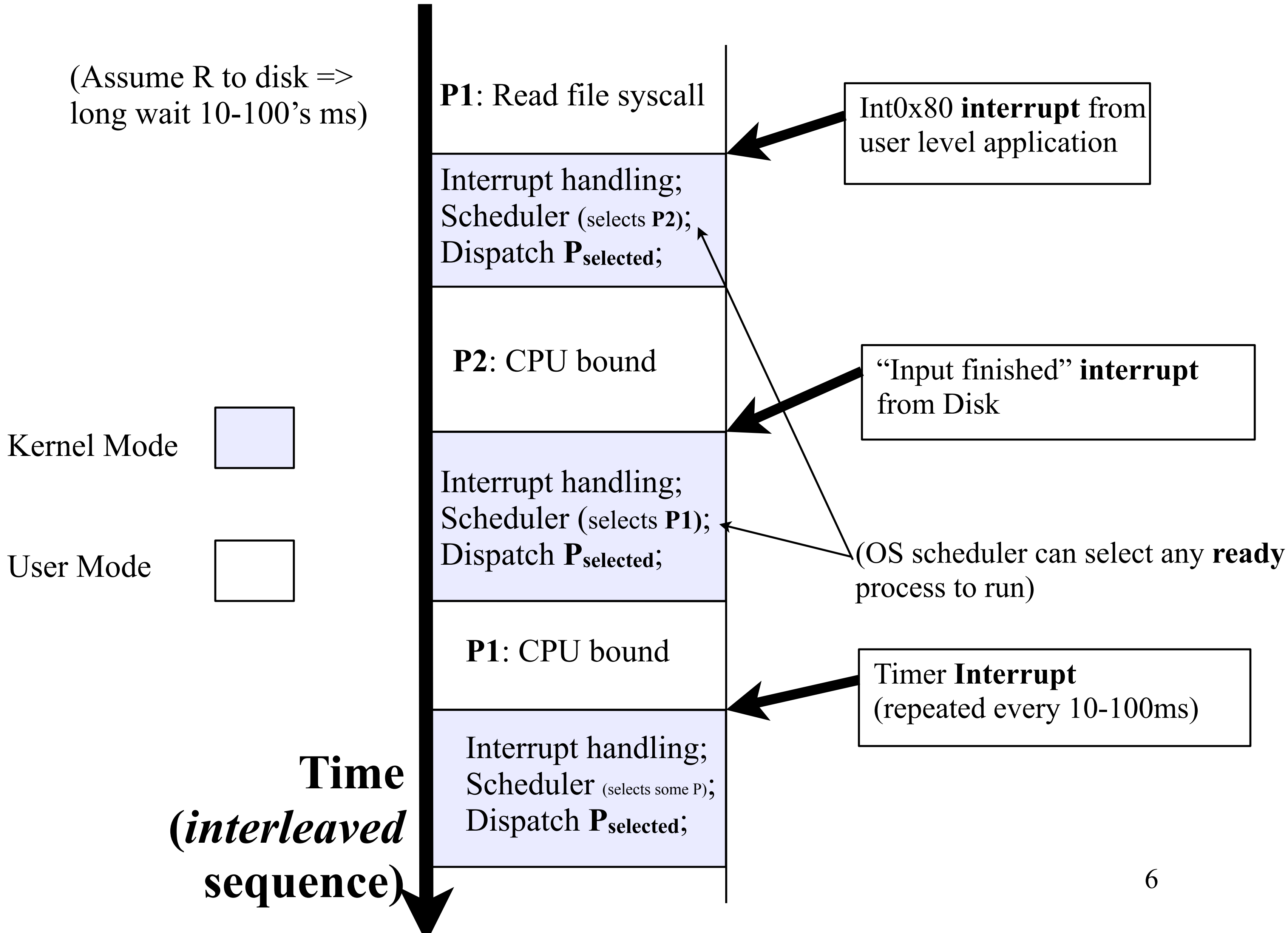


Process

- An instance of a program under execution
  - Program specifying (logical) control-flow (thread)
  - Data
  - Private address space
  - Open files
  - Run-time environment
- Very important operating system concept
  - Used for supporting the concurrent execution of independent or cooperating program instances
  - Used to structure applications and systems
  - Unit of Protection

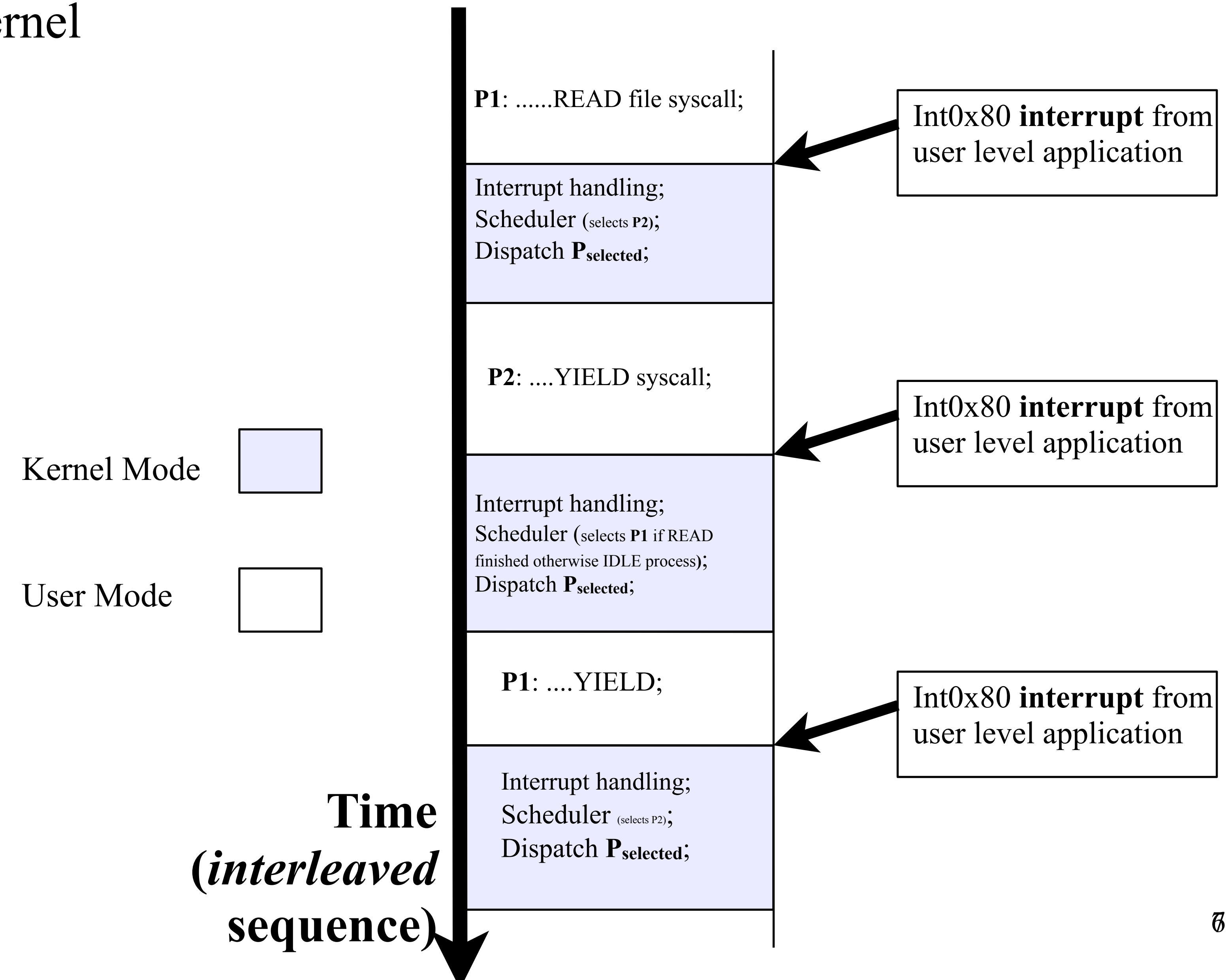
Flow of Execution

Two processes P1 and P2 executing *interleaved* on **Pre-Emptive** OS Kernel



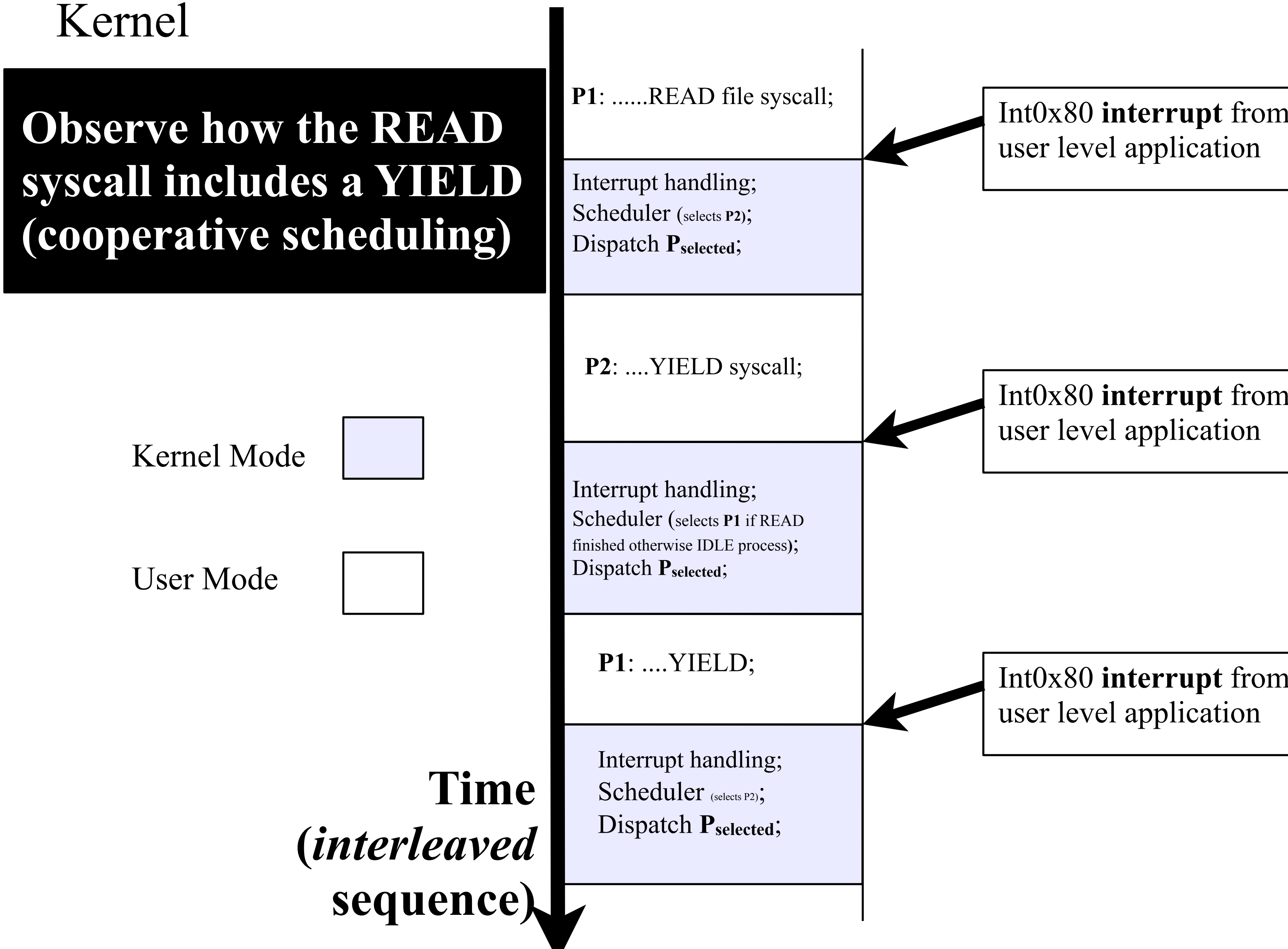
Flow of Execution

Two processes P1 and P2 executing *interleaved* on **Non-PreEmptive** OS Kernel



Flow of Execution

Two processes P1 and P2 executing *interleaved* on **Non-PreEmptive** OS Kernel



Concurrency & performance

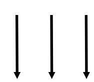
- Speedup
  - ideal: n processes, n speedup
  - reality: bottlenecks + overheads
    - Processes may have to be ordered for some operations, this will limit parallel pay-off
  - Questions
    - Speedup when
      - working with 1 partner?
      - working with 10 partners? 100? 1000? 10.000? ...
    - Give an example when we should benefit performance-wise *even on a single CPU with a single core?*
  - Also check out Amdahl's Law

Procedure, Co-routine, Thread, Process

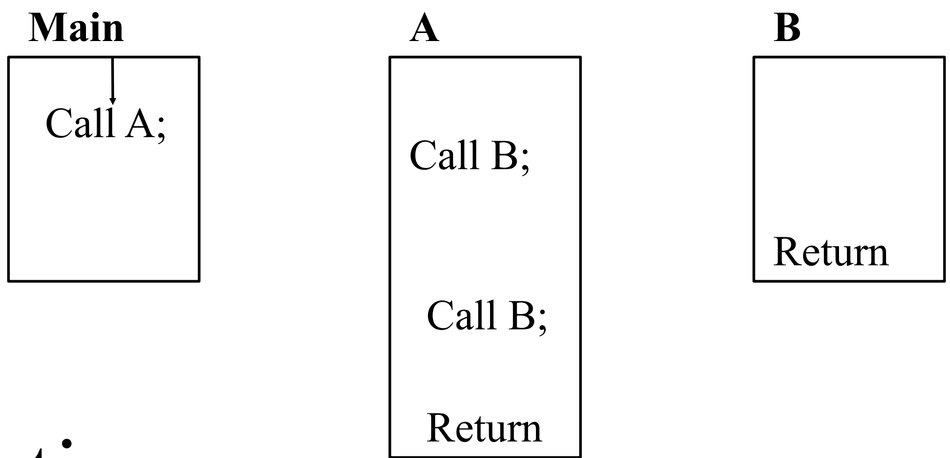
- Procedure, Function, (Sub)Routine
    - Call-execute all-return nesting
- Co-routine
    - Call-resumes-return

User level non preemptive "scheduler" spread "all over" user code

- Thread (more later)
- Process
  - Single threaded

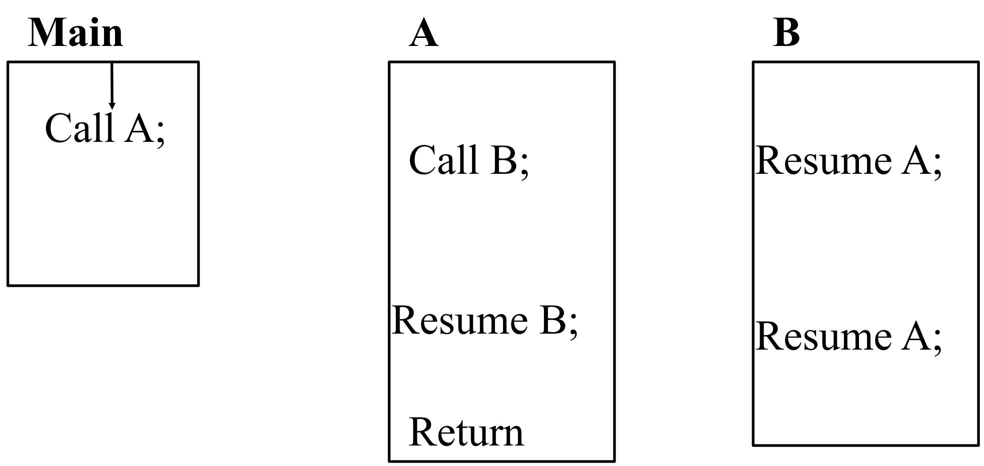
  - Multi threaded
  

Procedure, function, subroutine



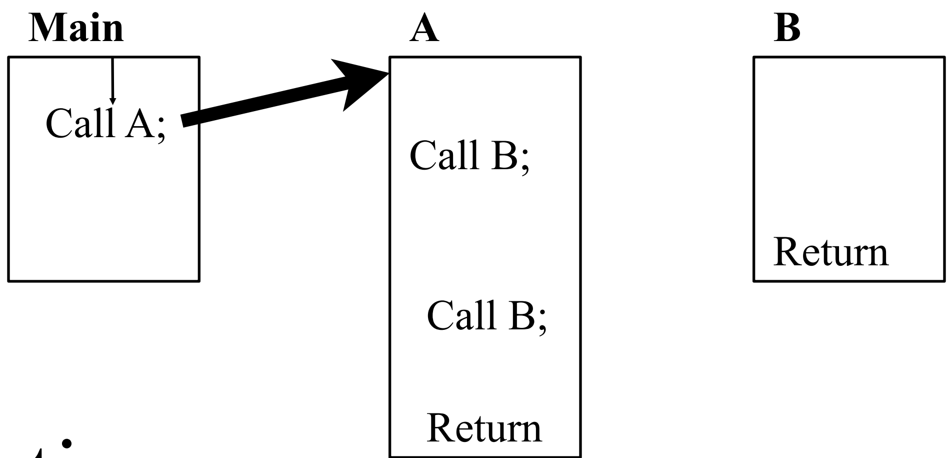
"Yield" when *finished*

Co-routine



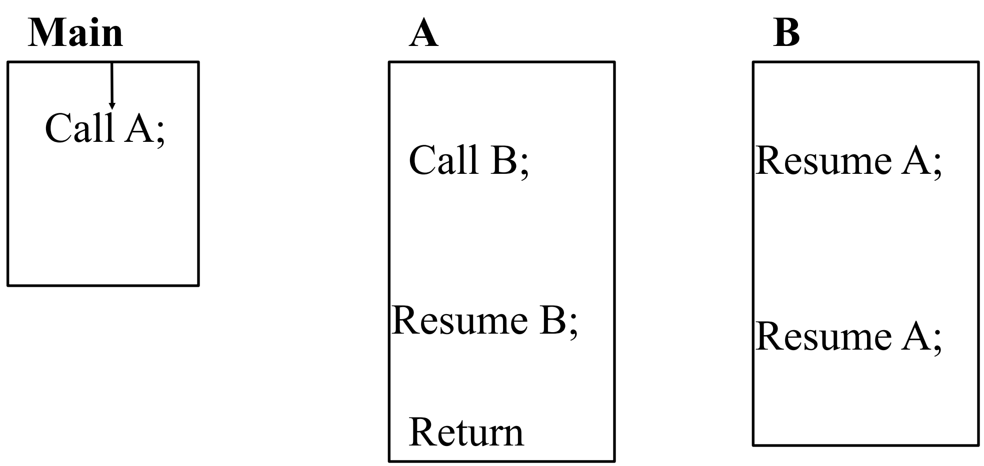
Yield *during* execution to share CPU

Procedure, function, subroutine



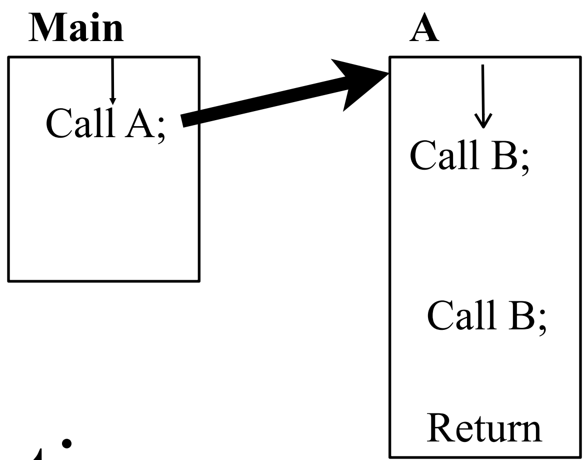
"Yield" when *finished*

Co-routine



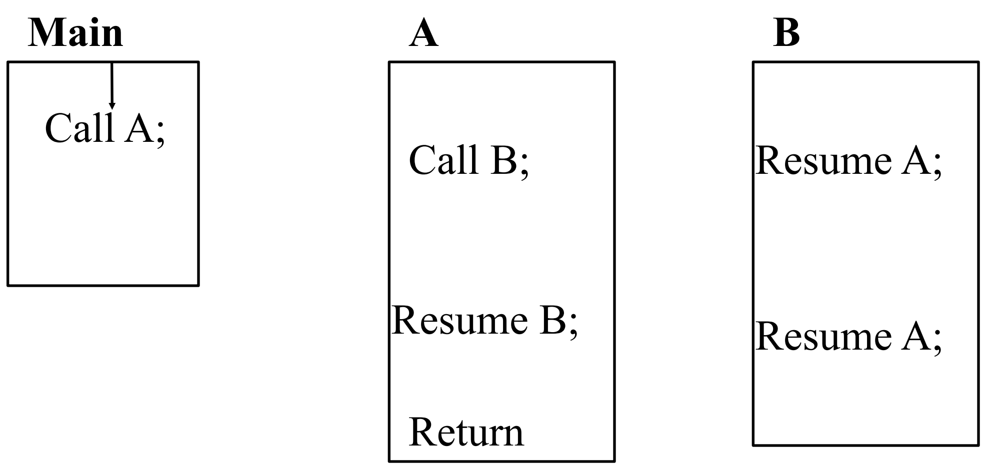
Yield *during* execution to share CPU

Procedure, function, subroutine



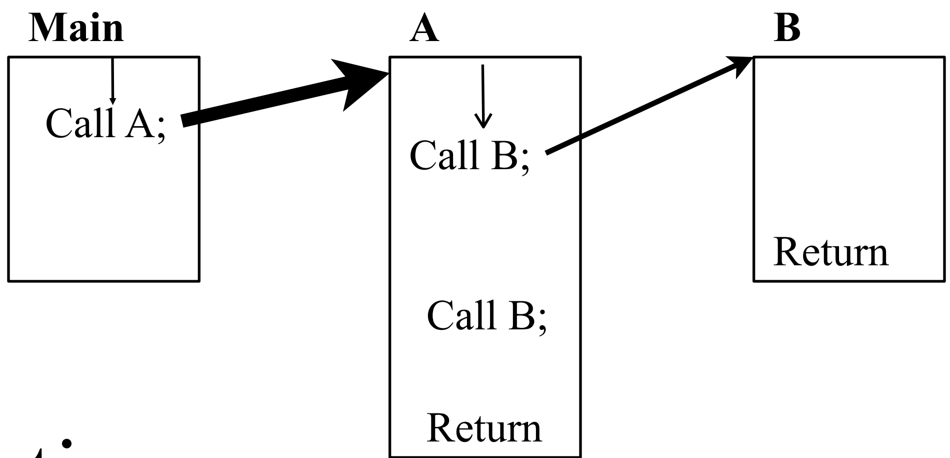
"Yield" when *finished*

Co-routine



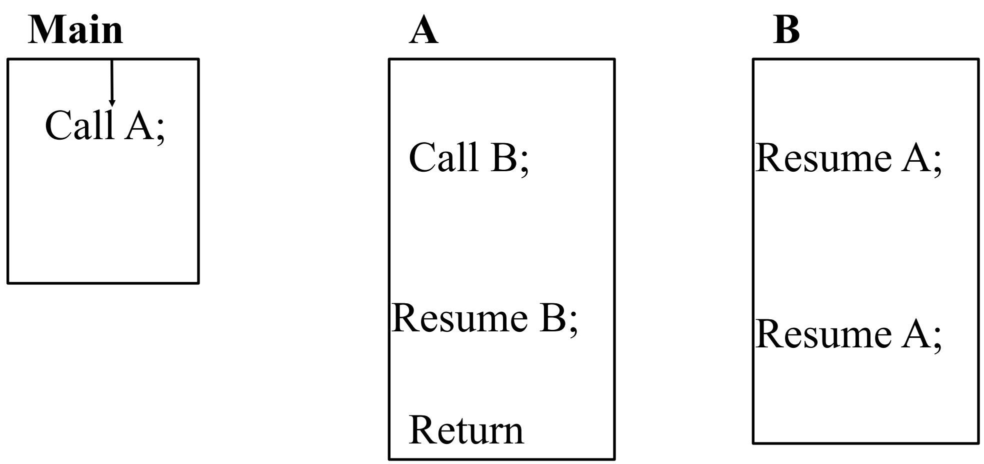
Yield *during* execution to share CPU

Procedure, function, subroutine



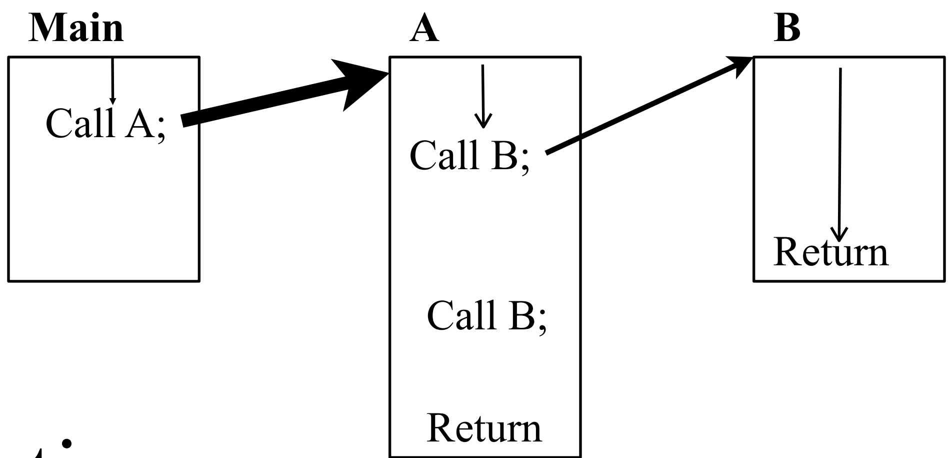
“Yield” when *finished*

Co-routine



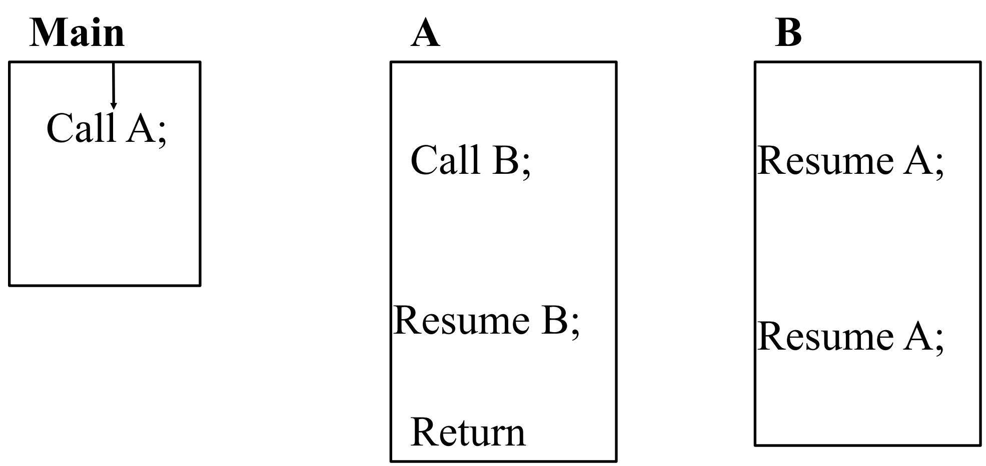
Yield *during* execution to share CPU

Procedure, function, subroutine



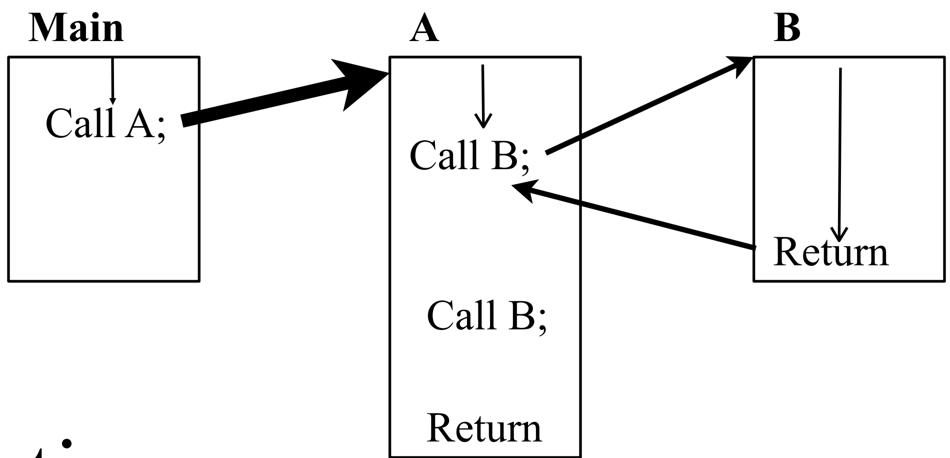
"Yield" when *finished*

Co-routine



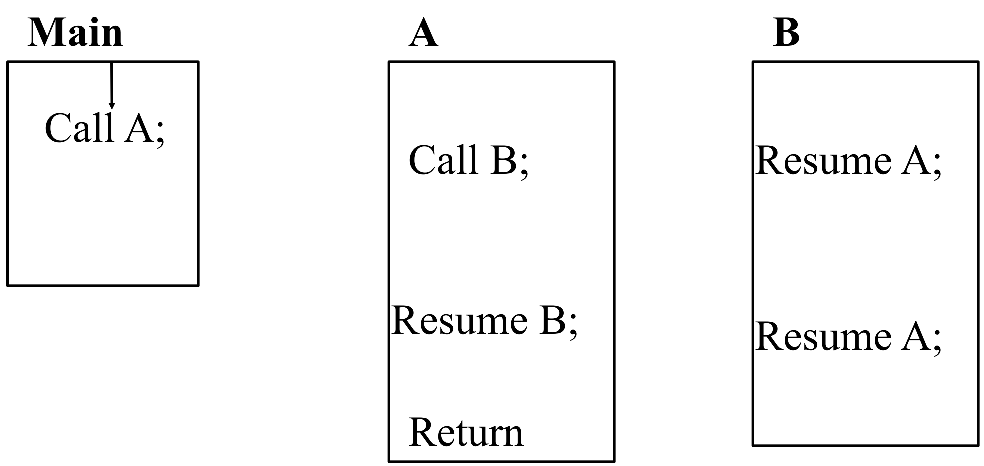
Yield *during* execution to share CPU

Procedure, function, subroutine



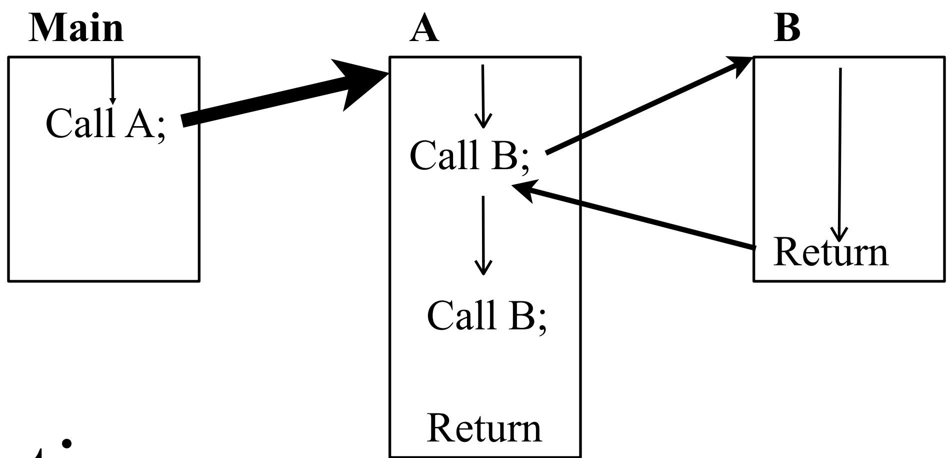
"Yield" when *finished*

Co-routine



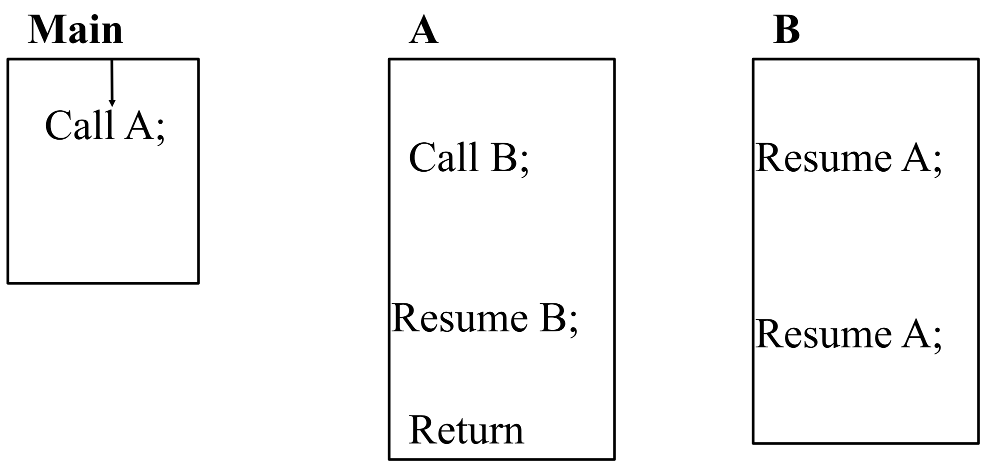
Yield *during* execution to share CPU

Procedure, function, subroutine



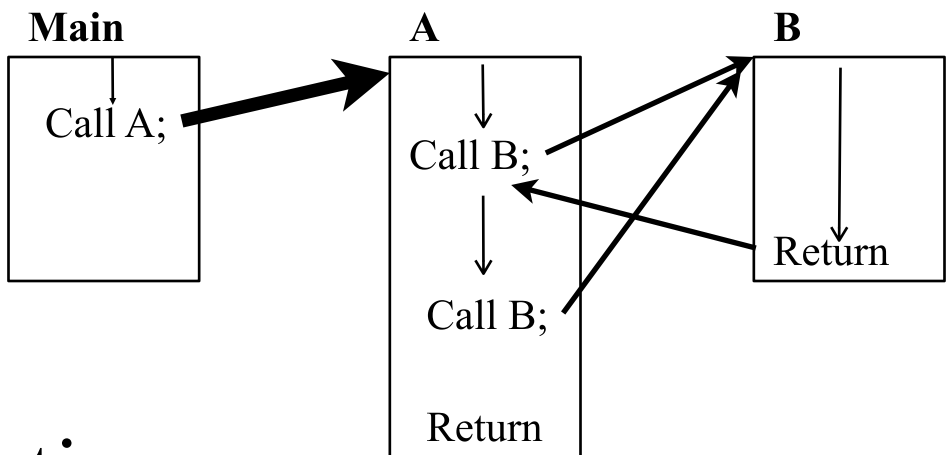
“Yield” when *finished*

Co-routine



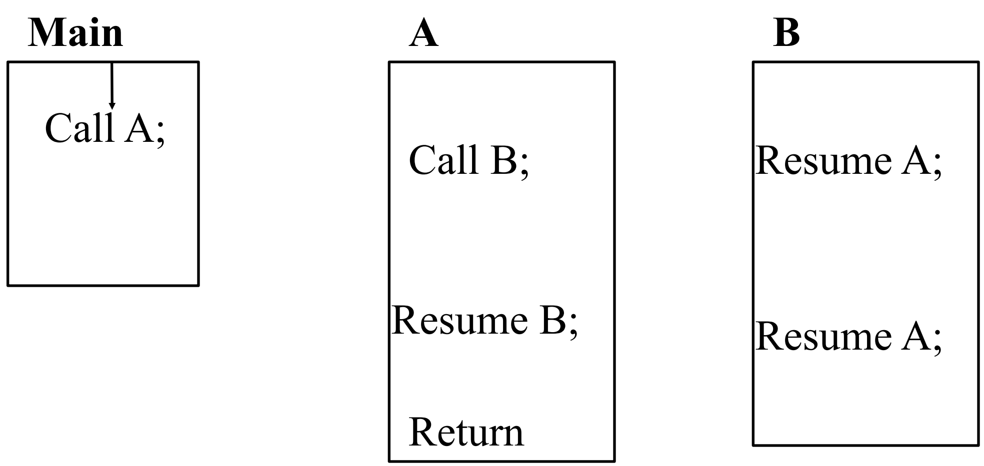
Yield *during* execution to share CPU

Procedure, function, subroutine



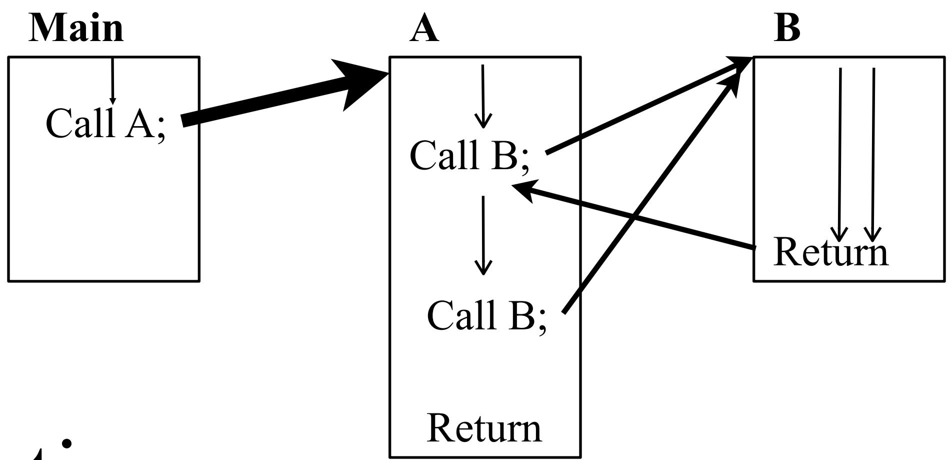
"Yield" when *finished*

Co-routine



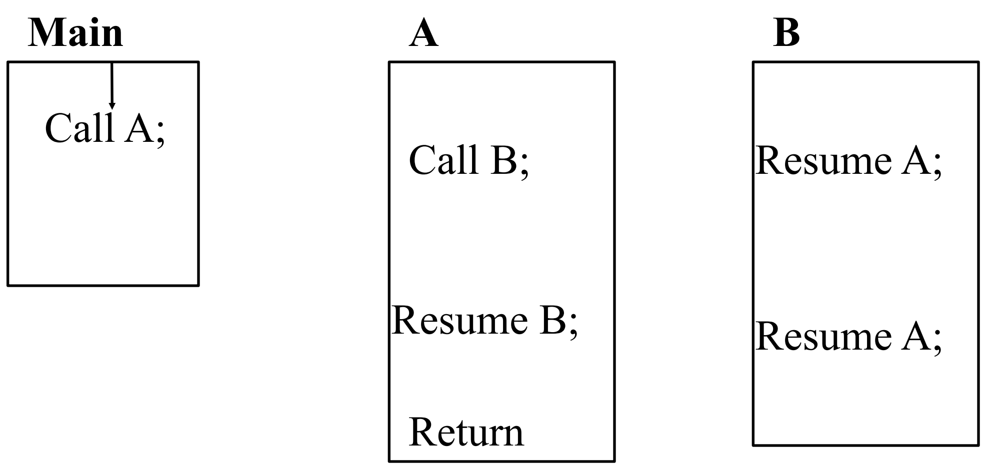
Yield *during* execution to share CPU

Procedure, function, subroutine



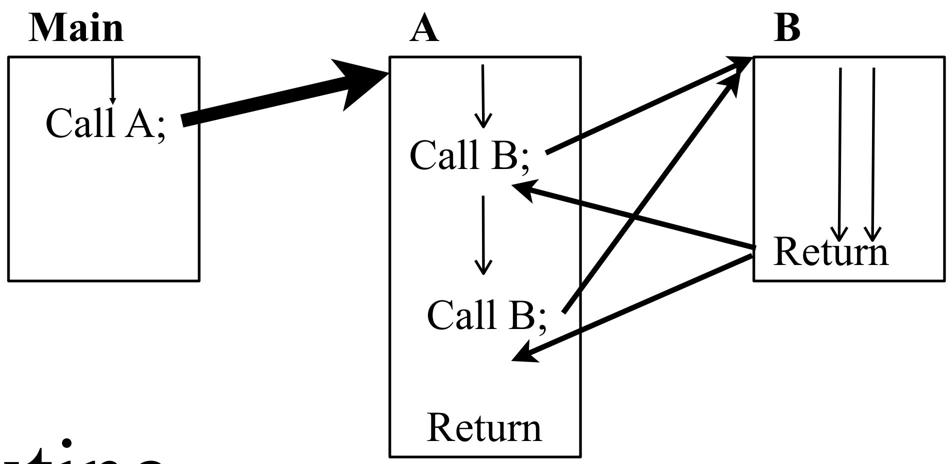
"Yield" when *finished*

Co-routine



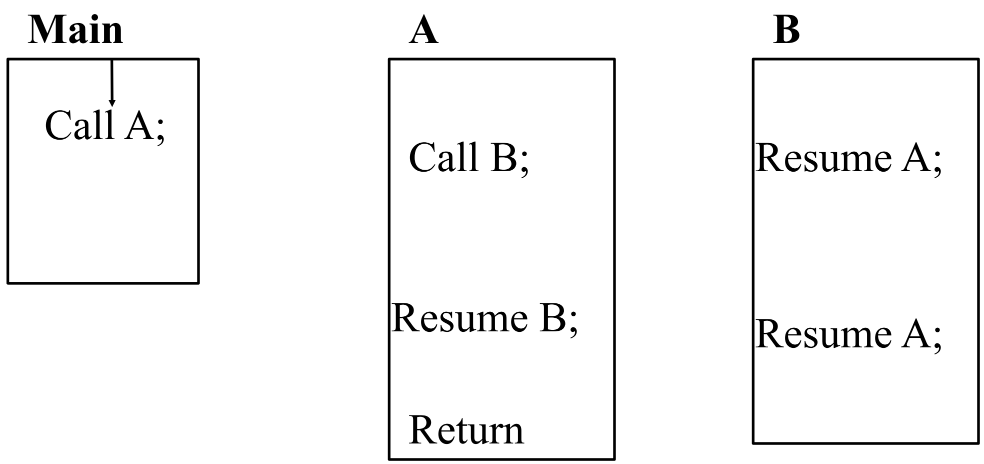
Yield *during* execution to share CPU

Procedure, function, subroutine



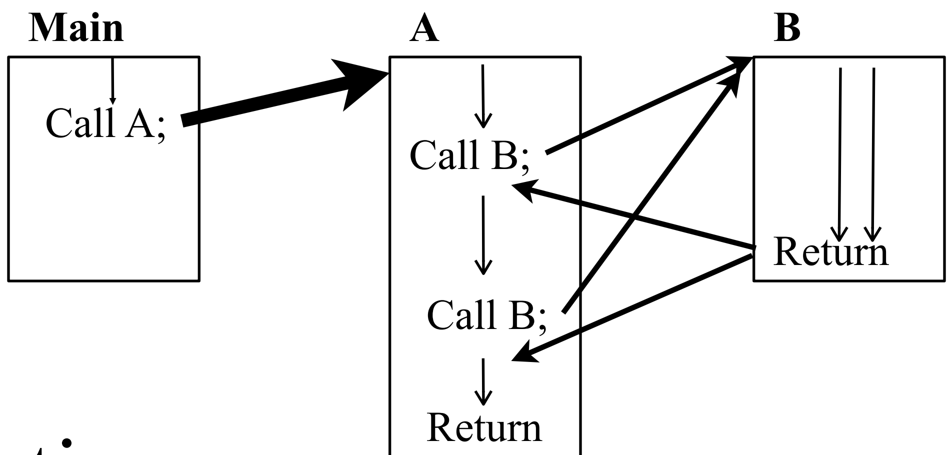
"Yield" when *finished*

Co-routine



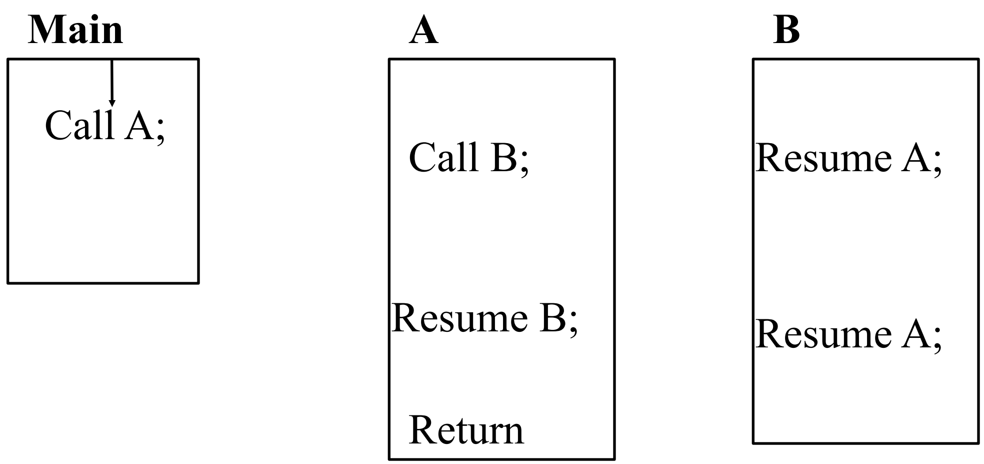
Yield *during* execution to share CPU

Procedure, function, subroutine



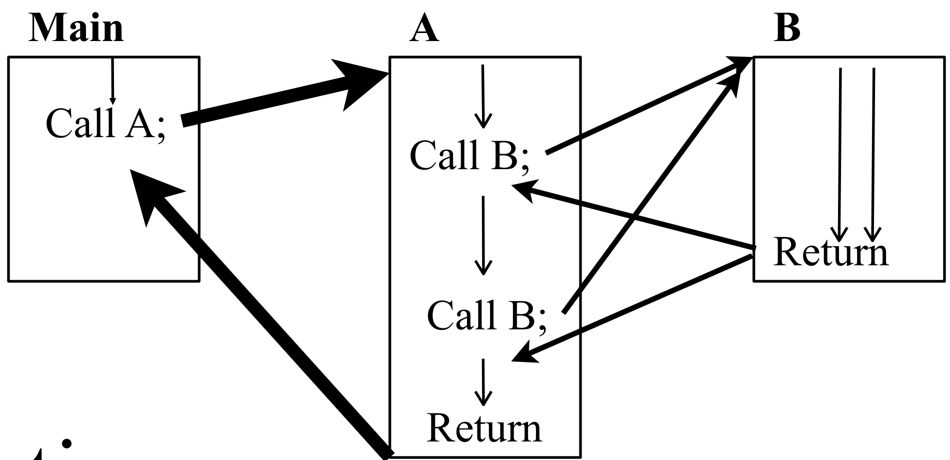
"Yield" when *finished*

Co-routine



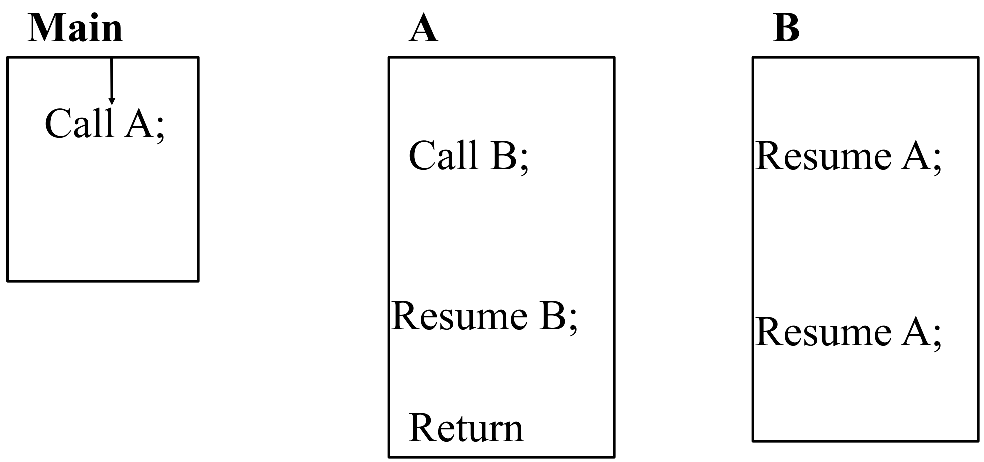
Yield *during* execution to share CPU

Procedure, function, subroutine



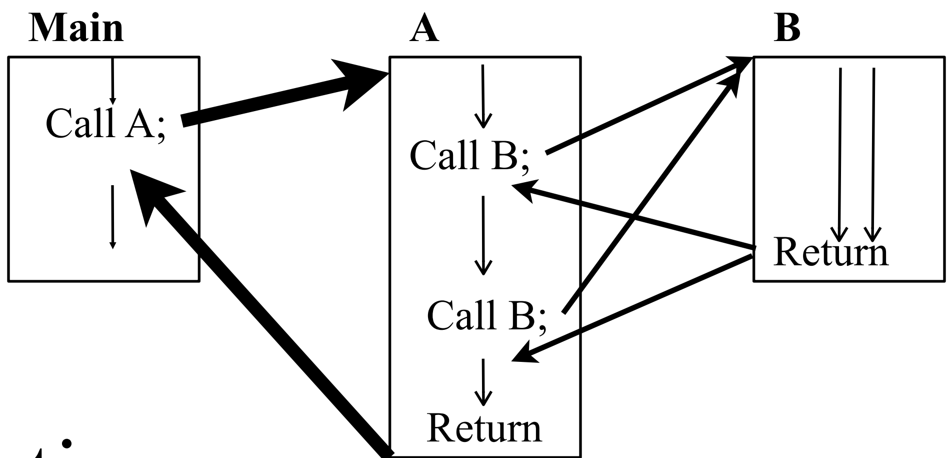
"Yield" when *finished*

Co-routine



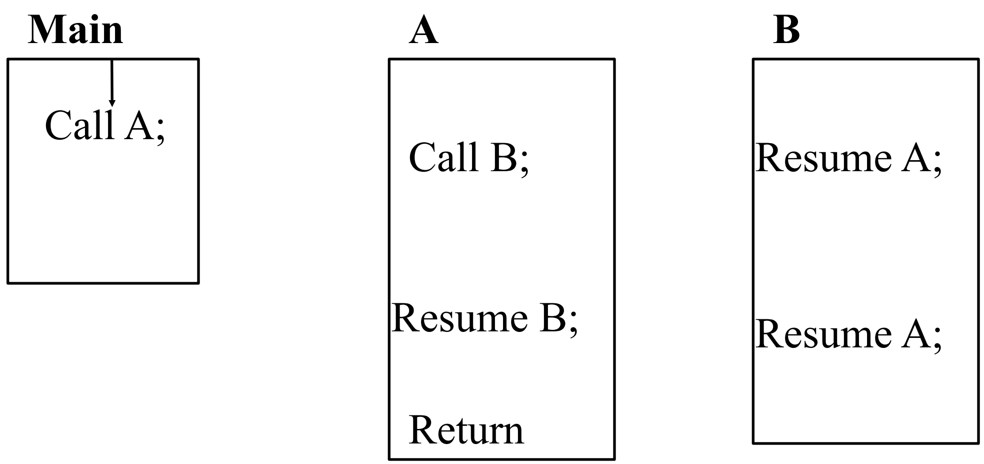
Yield *during* execution to share CPU

Procedure, function, subroutine



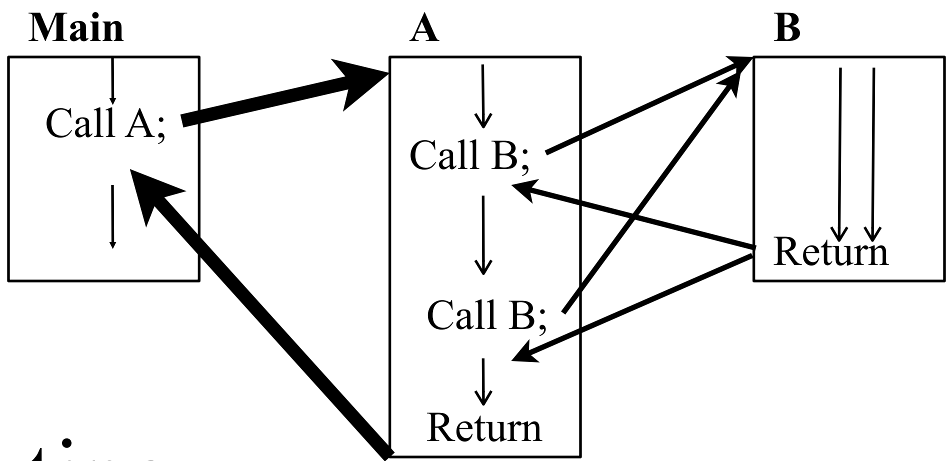
"Yield" when *finished*

Co-routine



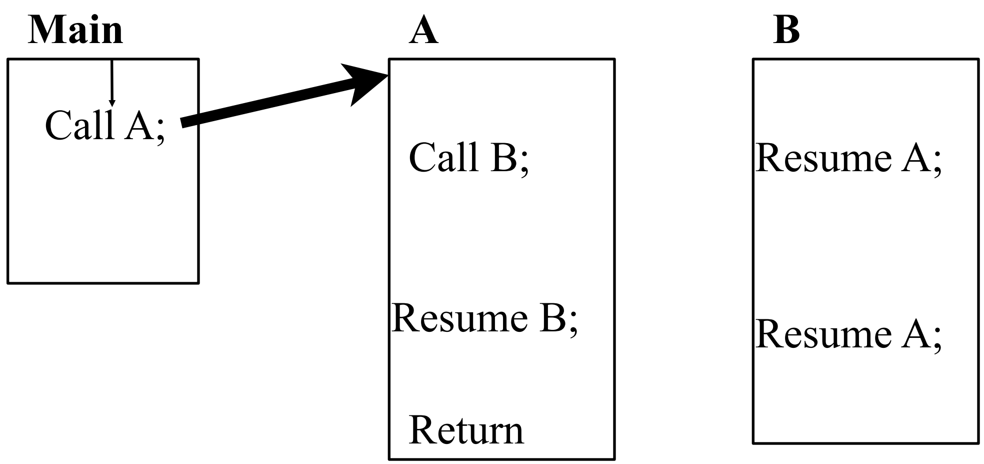
Yield *during* execution to share CPU

Procedure, function, subroutine



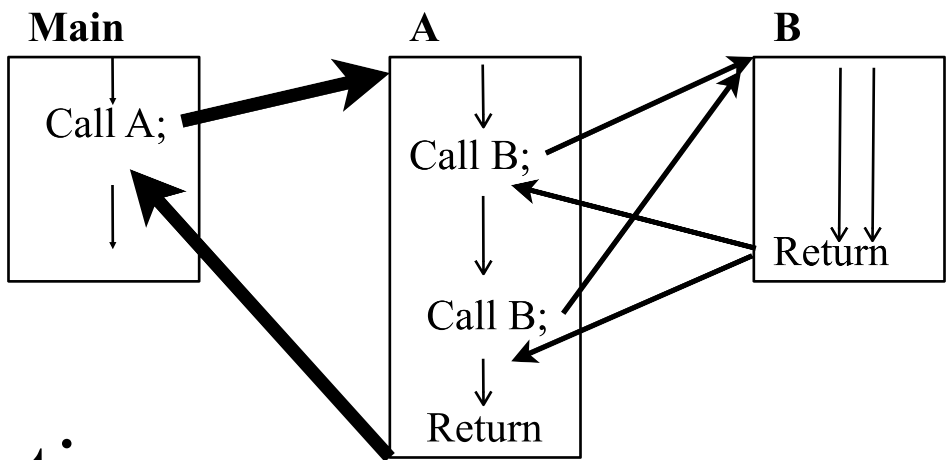
"Yield" when *finished*

Co-routine



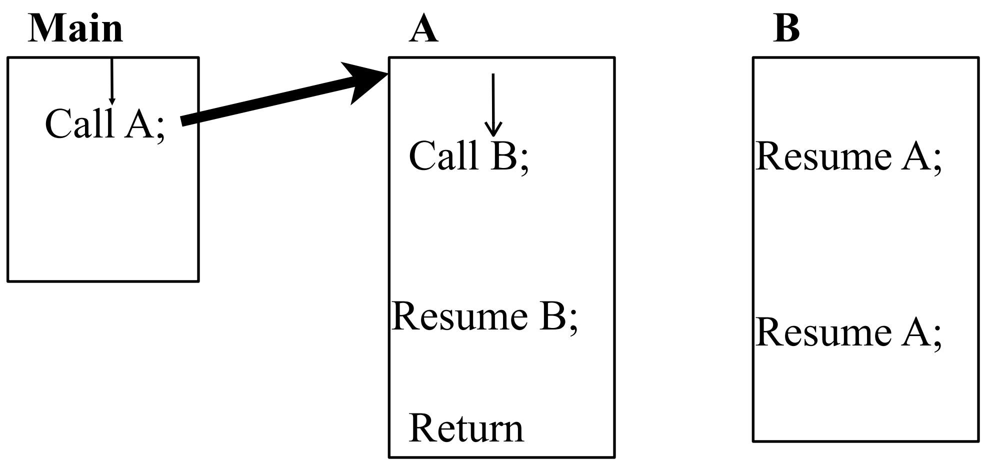
Yield *during* execution to share CPU

Procedure, function, subroutine



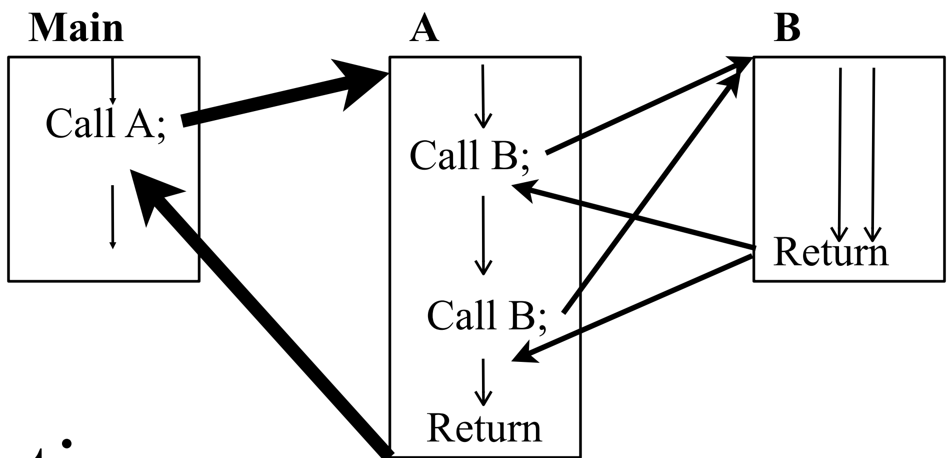
"Yield" when *finished*

Co-routine



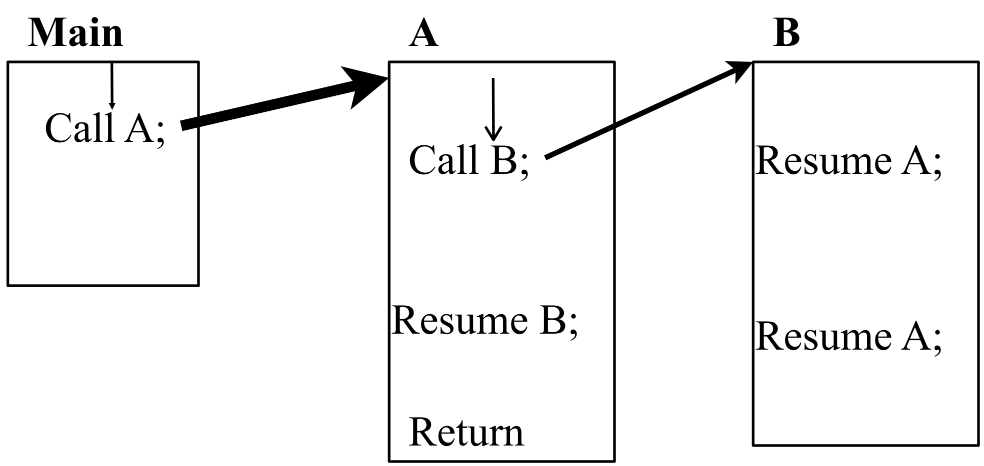
Yield *during* execution to share CPU

Procedure, function, subroutine



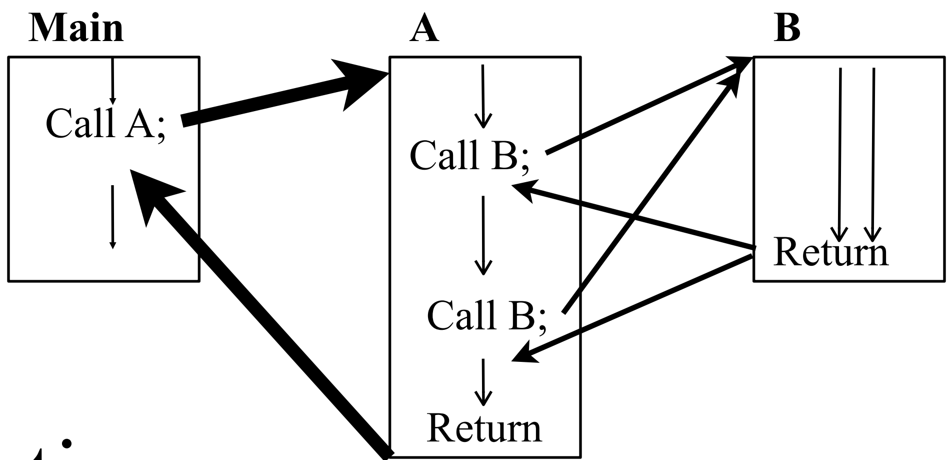
"Yield" when *finished*

Co-routine



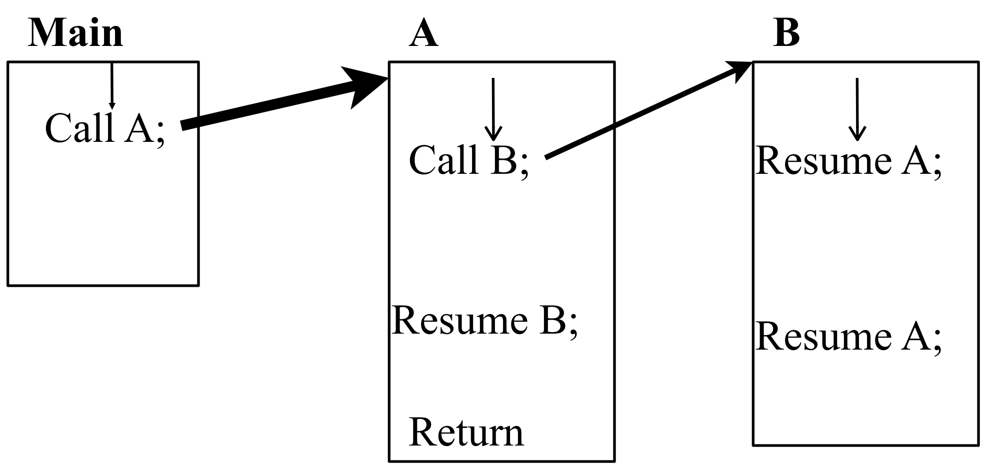
Yield *during* execution to share CPU

Procedure, function, subroutine



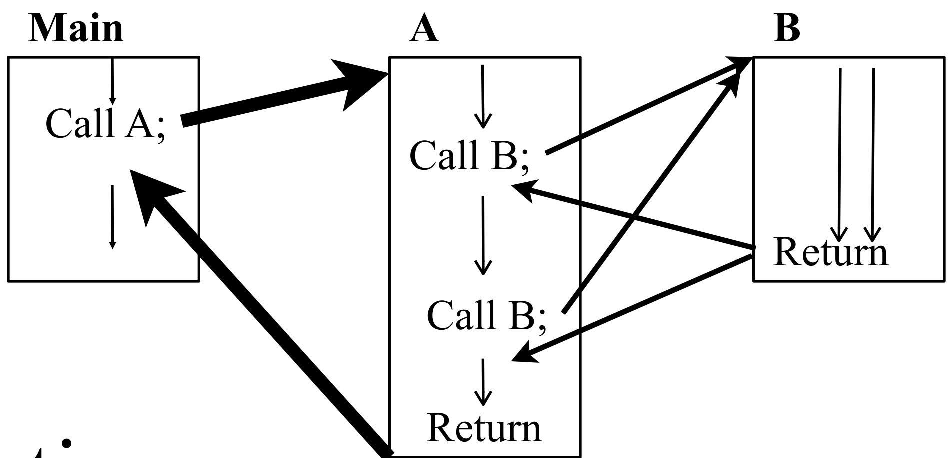
“Yield” when *finished*

Co-routine



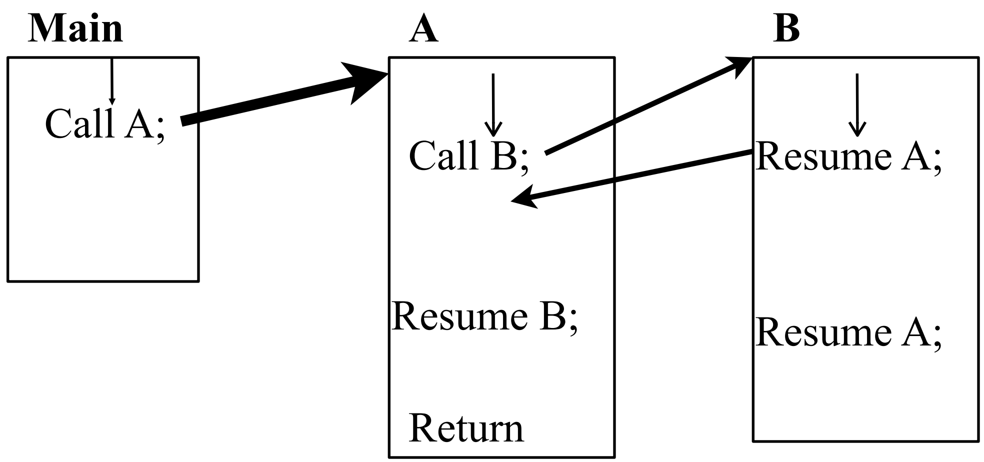
Yield *during* execution to share CPU

Procedure, function, subroutine



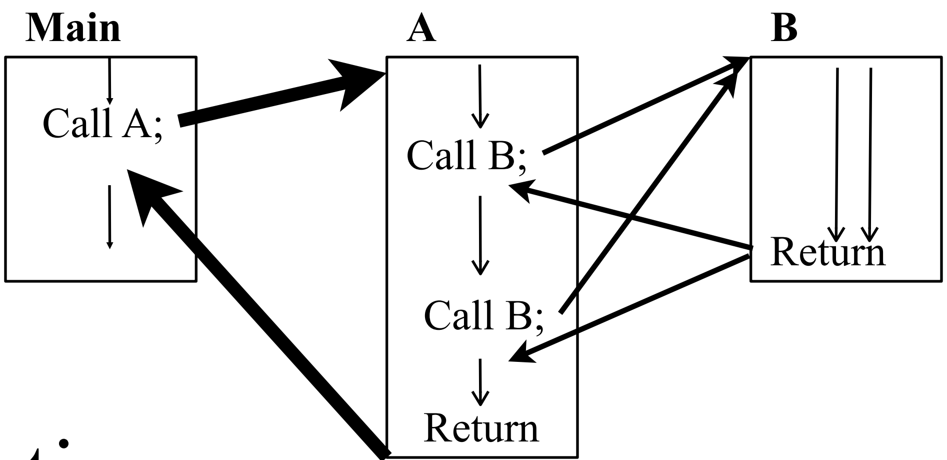
"Yield" when *finished*

Co-routine



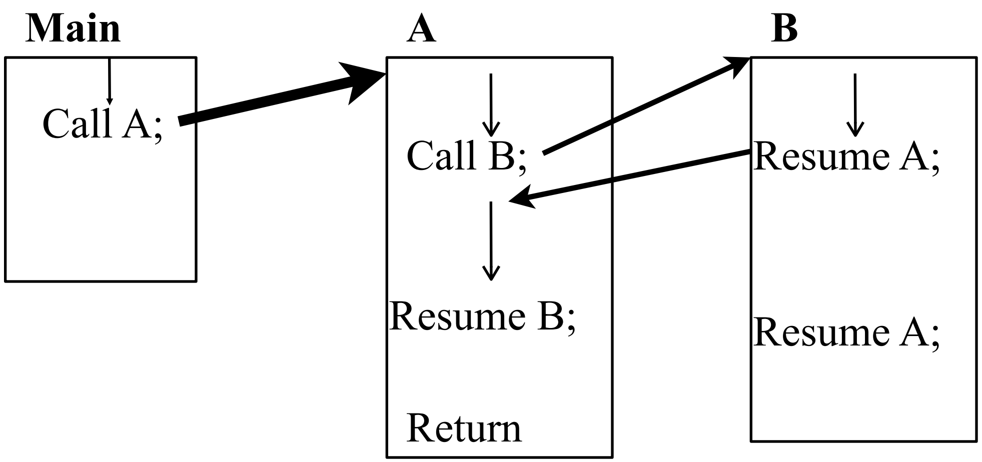
Yield *during* execution to share CPU

Procedure, function, subroutine



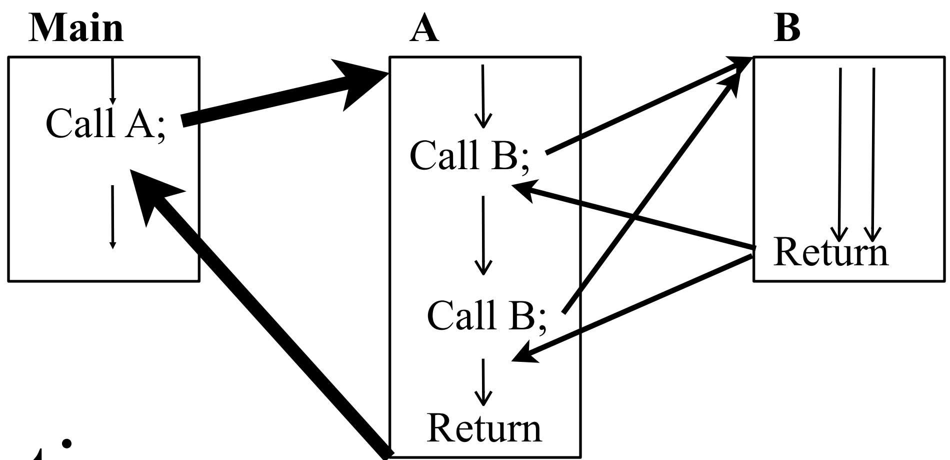
"Yield" when *finished*

Co-routine



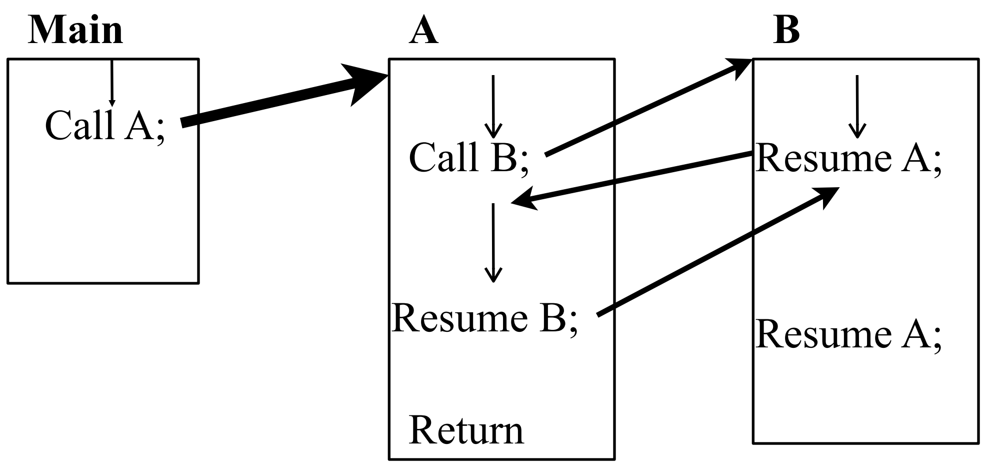
Yield *during* execution to share CPU

Procedure, function, subroutine



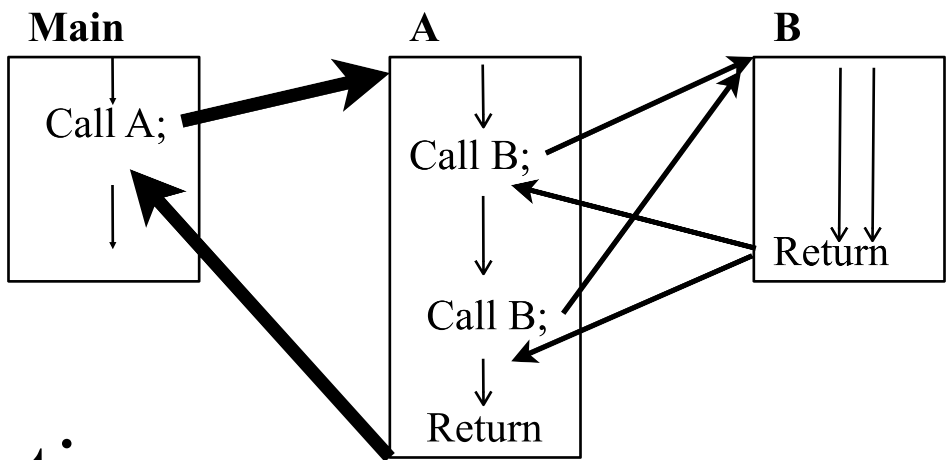
"Yield" when *finished*

Co-routine



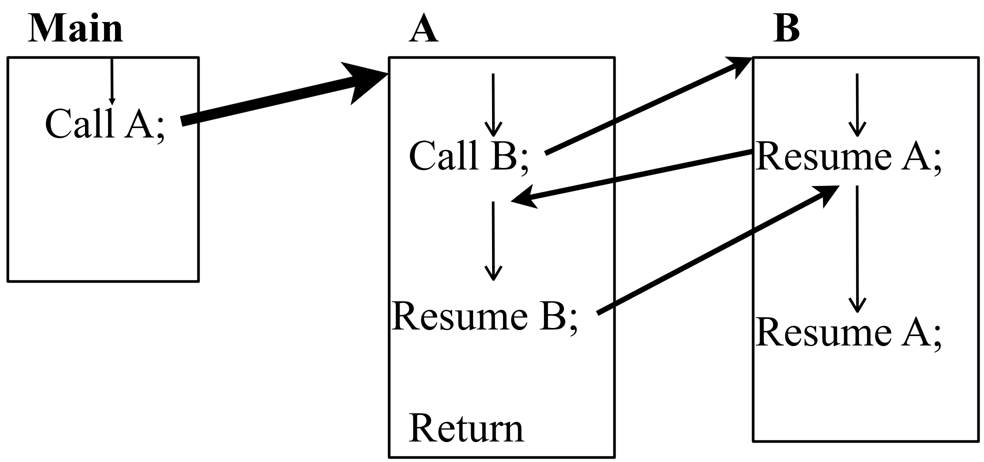
Yield *during* execution to share CPU

Procedure, function, subroutine



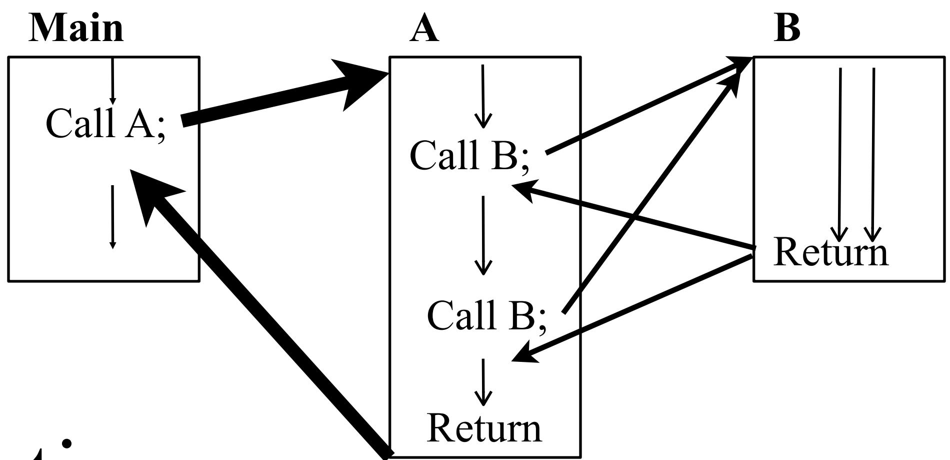
"Yield" when *finished*

Co-routine



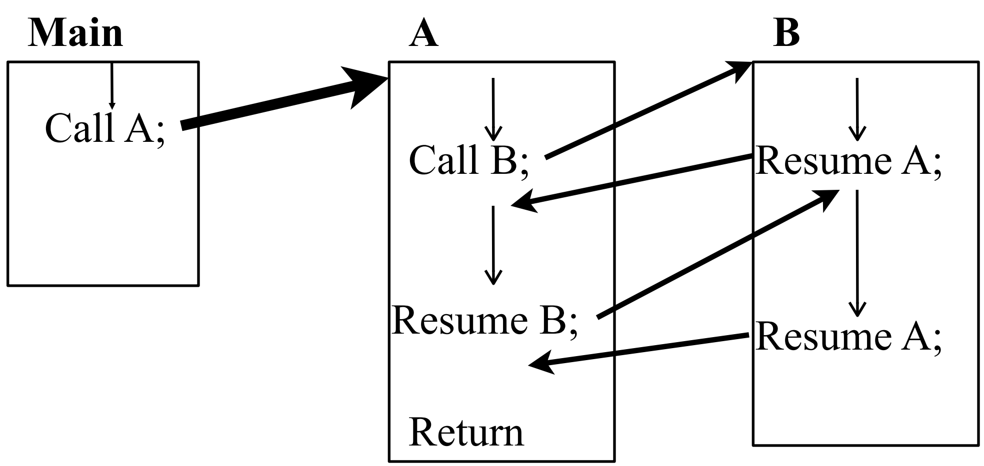
Yield *during* execution to share CPU

Procedure, function, subroutine



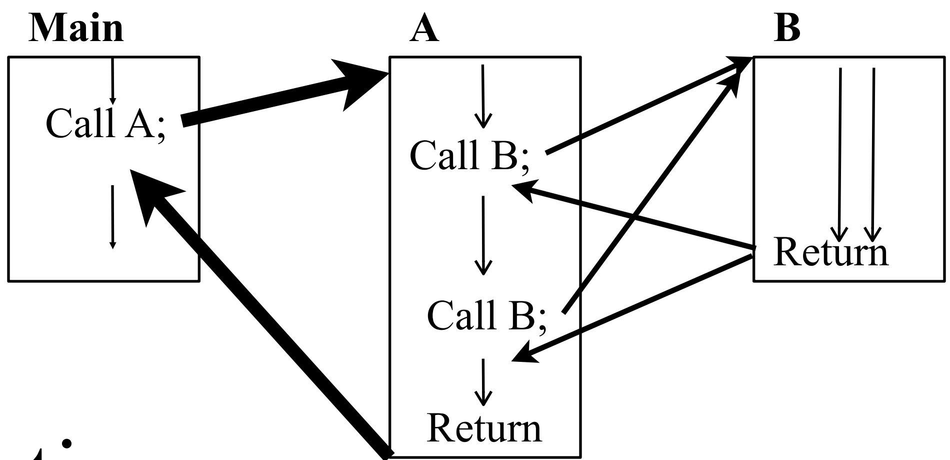
"Yield" when *finished*

Co-routine



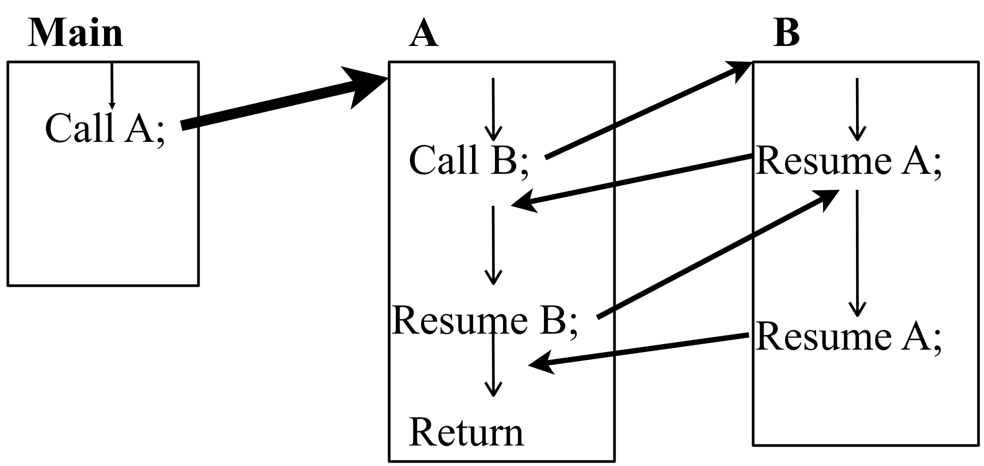
Yield *during* execution to share CPU

Procedure, function, subroutine



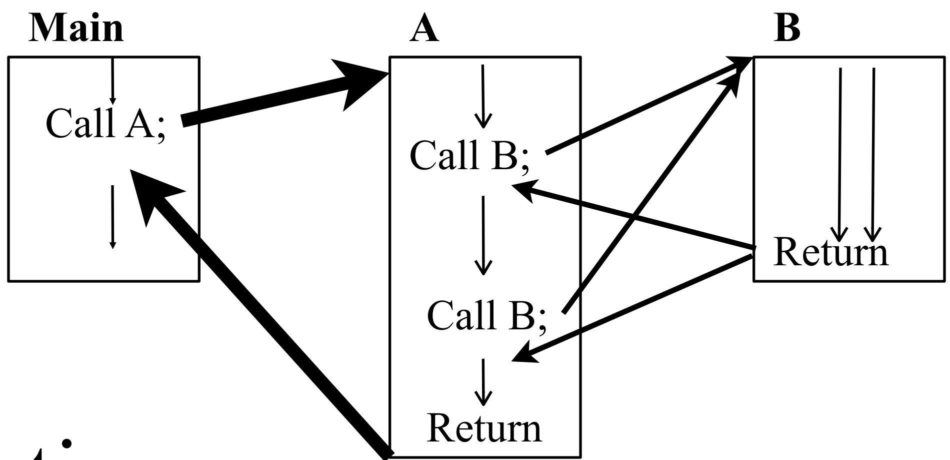
"Yield" when *finished*

Co-routine



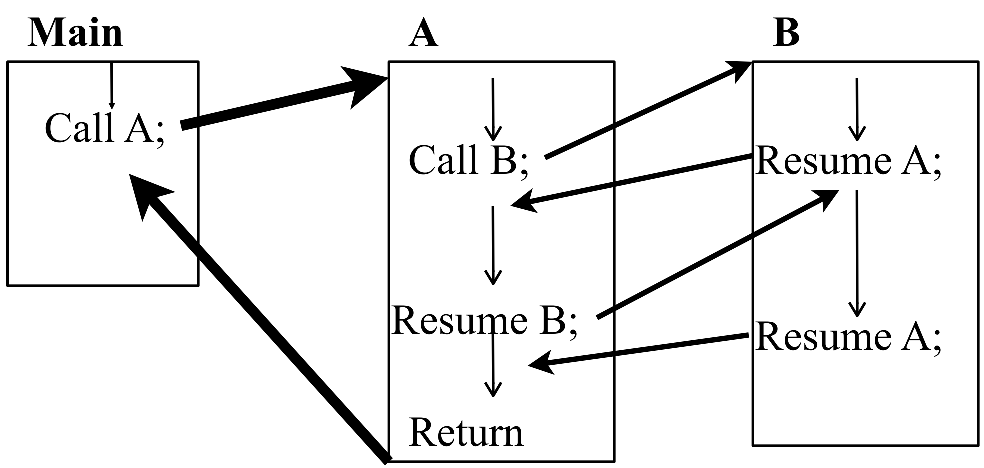
Yield *during* execution to share CPU

Procedure, function, subroutine



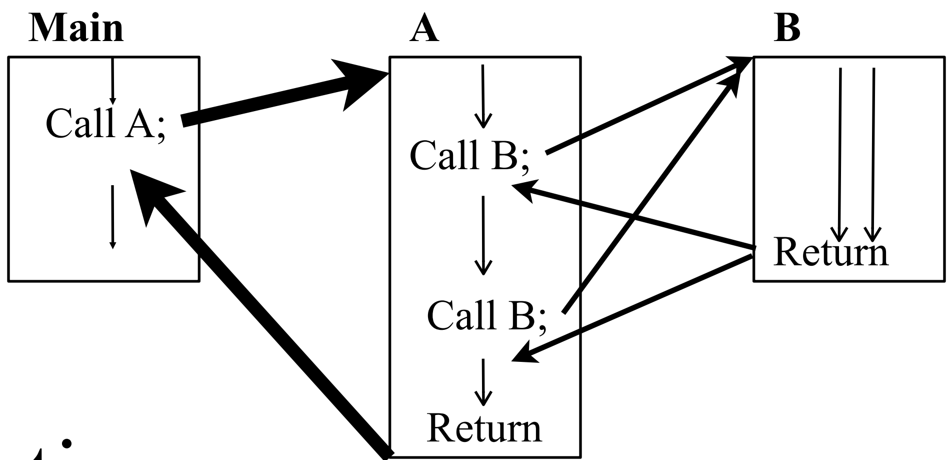
"Yield" when *finished*

Co-routine



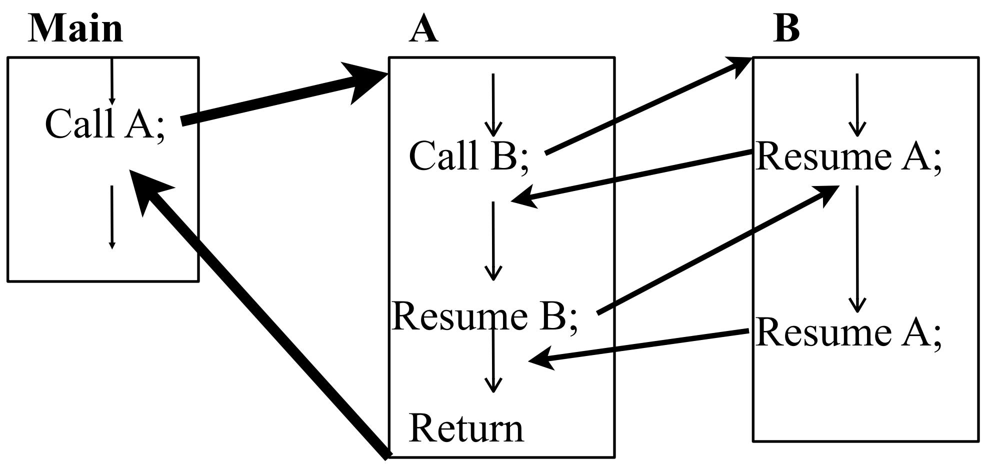
Yield *during* execution to share CPU

Procedure, function, subroutine



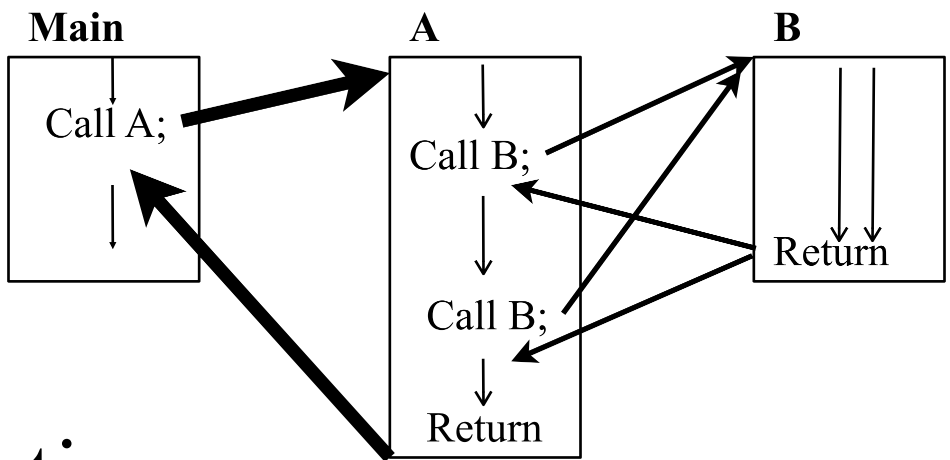
"Yield" when *finished*

Co-routine



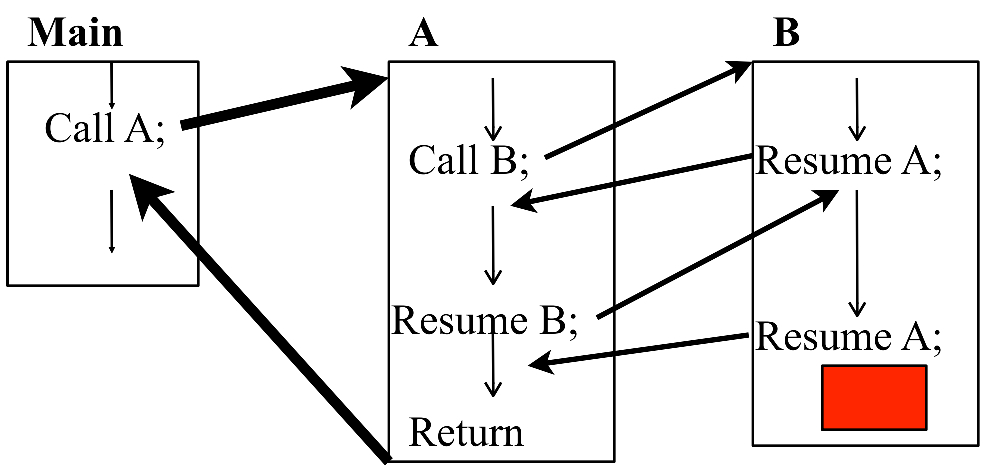
Yield *during* execution to share CPU

Procedure, function, subroutine



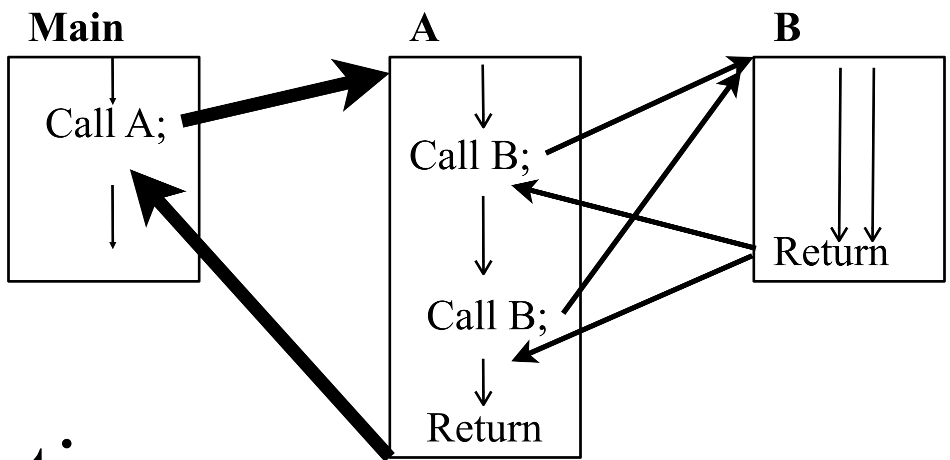
"Yield" when *finished*

Co-routine



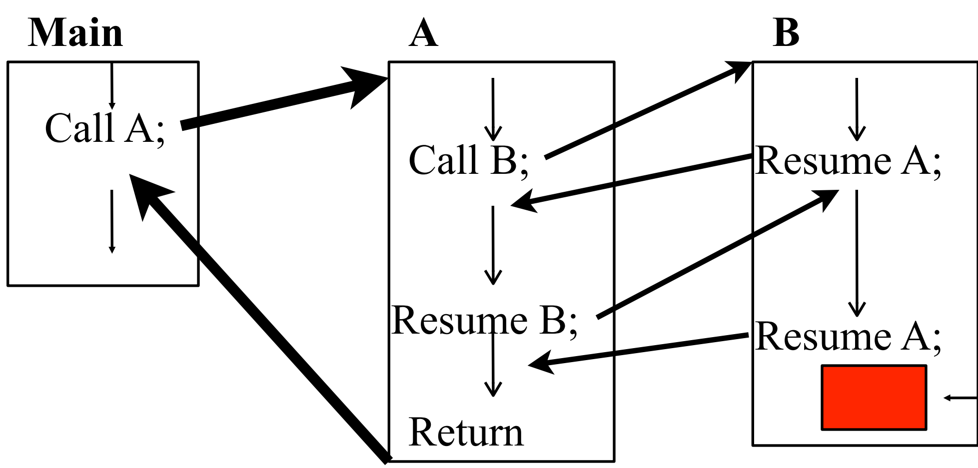
Yield *during* execution to share CPU

Procedure, function, subroutine



"Yield" when *finished*

Co-routine



Yield *during* execution to share CPU

Never executed

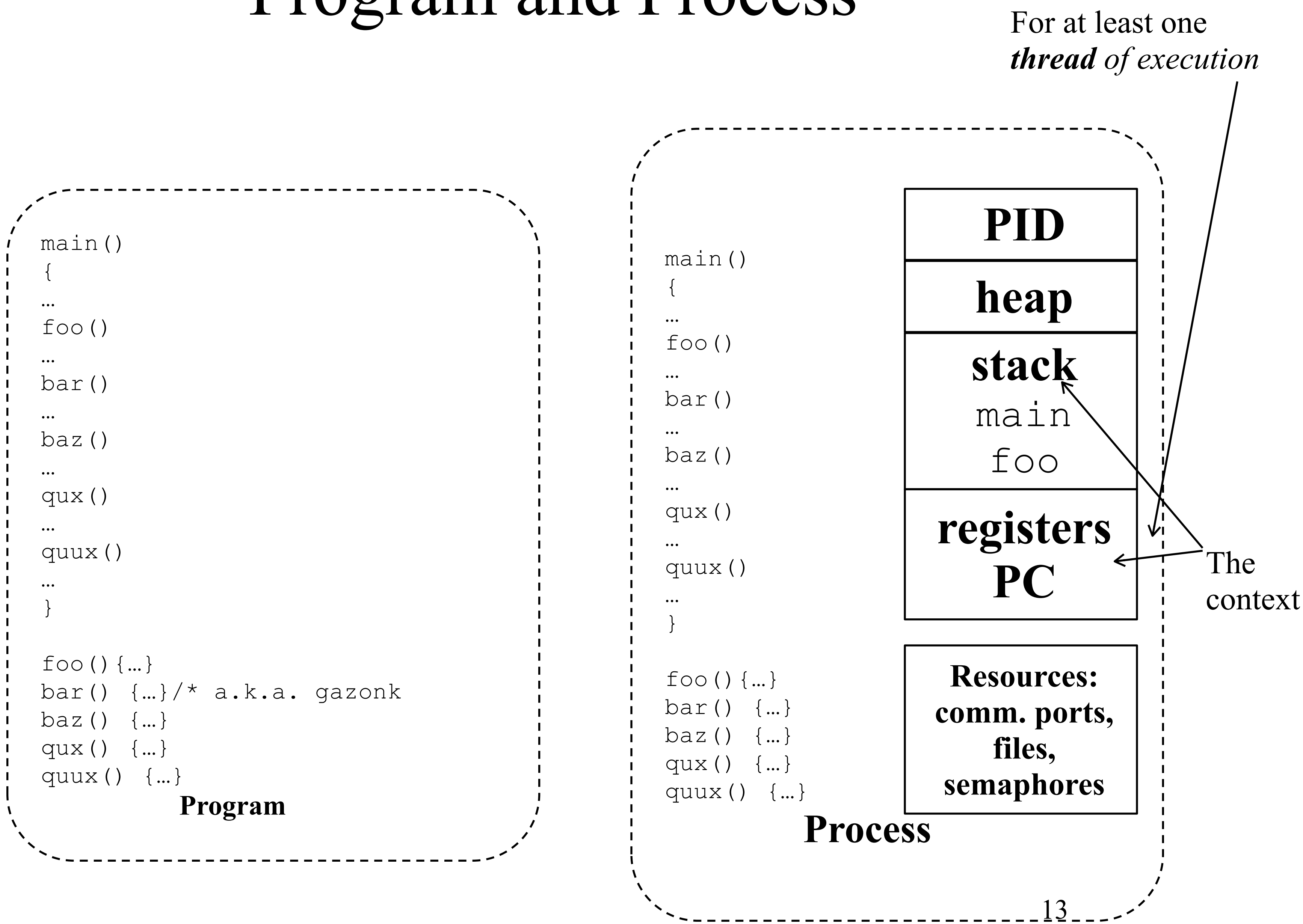
Process

- "Modern" process: **Process** and **Thread** are separated as concepts
- Process—Unit of Resource Allocation—Defines the context
- Thread—Control Thread—Unit of execution, scheduling
- Every process has at least one thread

Single threaded sequential Process

- Sequential execution of operations
  - No concurrency inside a (**single** threaded) process
  - Everything happens sequentially
- Process state defined by:
  - Registers
  - Stack(s)
  - Main memory
  - I/O devices
    - Files and their state
    - Communication ports
  - Other resources

Program and Process

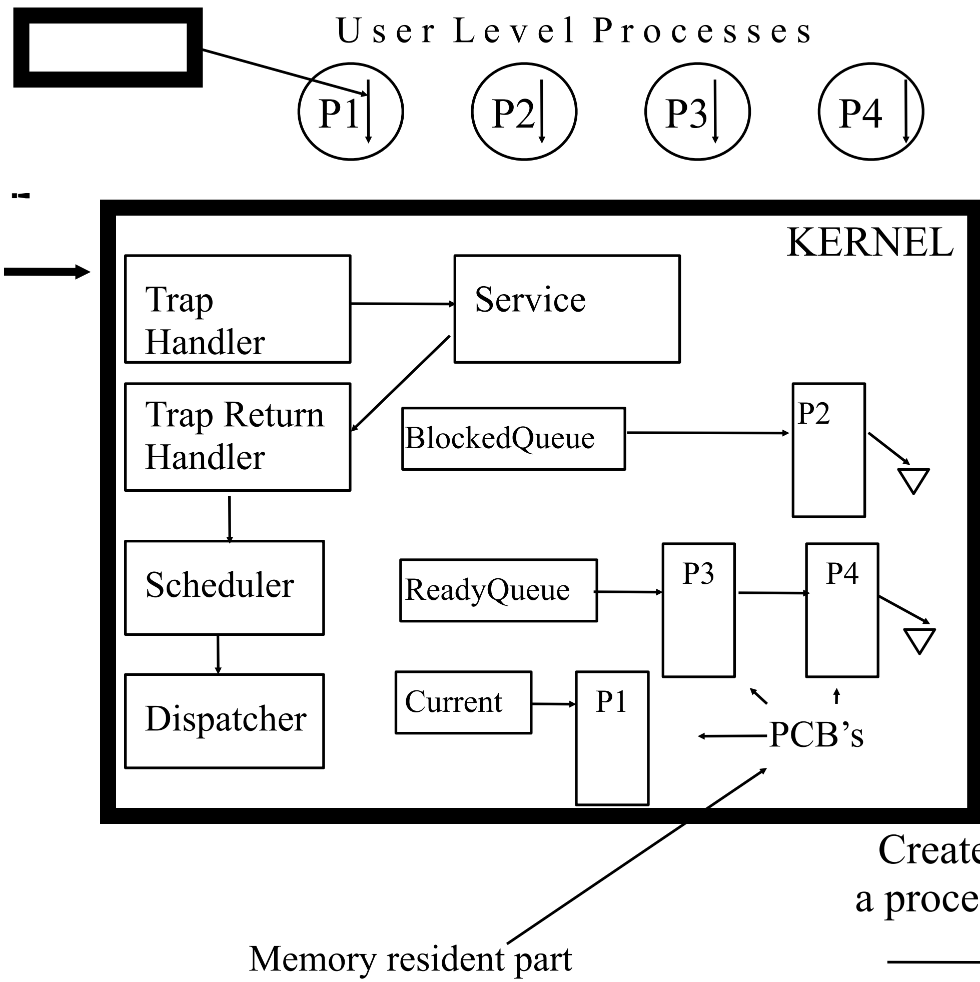


Process vs. Program

- Process “>” program
  - Program is just part of process state
  - Example: many users can run the same program
- Process “<“ program
  - A program can invoke more than one process
  - Example: Fork off processes

Process State Transitions

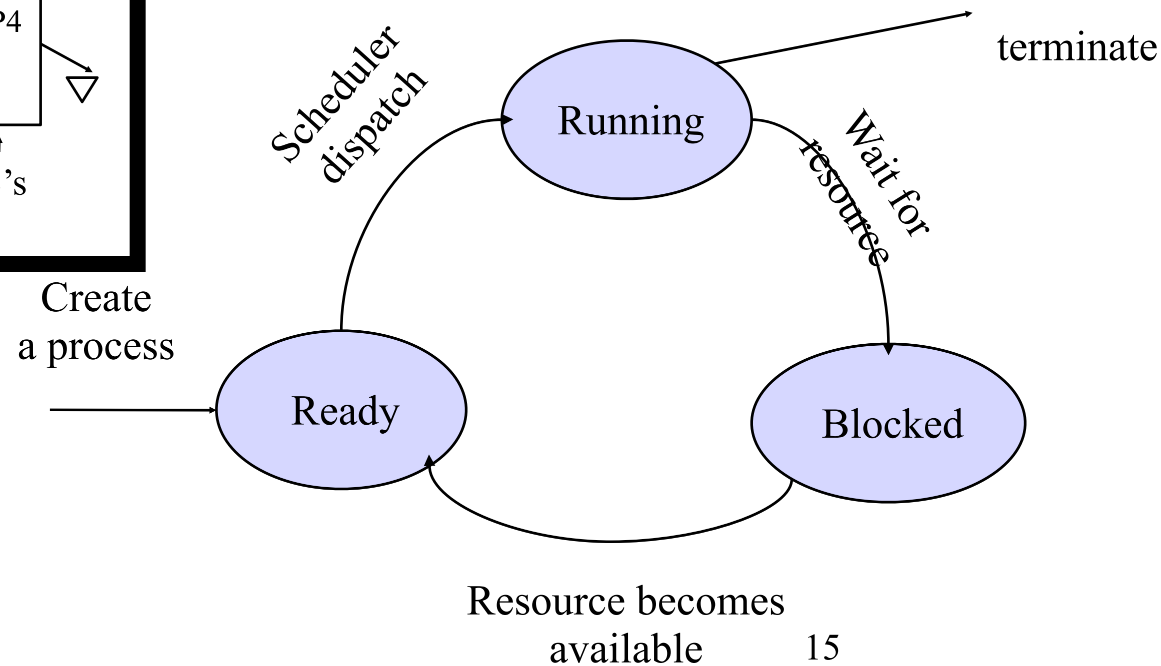
Instruction Pointer (program counter) in the EIP register



MULTIPROGRAMMING

•Uniprocessor: *Interleaving* ("pseudoparallelism")

•Multiprocessor: *Overlapping* ("true paralellism")



What needs to be saved and restored on a context switch?

- Volatile state
  - Program counter (Program Counter (PC) also called Instruction Pointer (Intel: EIP))
  - Processor status register
  - Other register contents
  - User and kernel stack pointers
  - A pointer to the address space in which the process runs
    - the process's page table directory

Basic Flow of Context Switch

Basic Flow of Context Switch

- **Save**(volatile machine state, current process);
  - done by HW and Interrupt handler

Basic Flow of Context Switch

- **Save**(volatile machine state, current process);
  - done by HW and Interrupt handler
- **Load**(another process's saved volatile state);
  - selecting another process is done by OS Kernel Scheduler
  - loading is done by OS Kernel Dispatcher

Basic Flow of Context Switch

- **Save**(volatile machine state, current process);
  - done by HW and Interrupt handler
- **Load**(another process's saved volatile state);
  - selecting another process is done by OS Kernel Scheduler
  - loading is done by OS Kernel Dispatcher
- **Start**(new process);
  - done by OS Kernel Dispatcher

Implementing processes

- OS (kernel) needs to keep track of all processes
  - Progress
  - Metadata (priorities etc.) used by OS
  - Memory
  - Files
  - State including waiting for conditions, signals, and messages
- Process table with one entry (Process Control Block) per process
- Will also have the processes in *queues*

Make a Process

- Creation
  - load code and data into memory
  - create an empty stack
  - initialize state to same as after a process switch
  - make process READY to run
    - insert into OS scheduler queue (Ready_Queue)
- Clone
  - Stop *current* process and save (its) state
  - make copy of *current*s code, data, stack and OS state
  - make the new process READY to run

Process Control Block (PCB)

- Process management info
  - State (ready, running, blocked)
  - Registers, PSW, EFLAGS, and other CPU state
  - Stack, code, and data segment
- Memory management info
  - Segments, page table, stats, etc
- I/O and file management
  - Communication ports, directories, file descriptors, etc.
- *OS must allocate resources to each process, and do the state transitions*

Where Should PCB Be Kept?

- Save the PCB on user stack
  - Many processors have a special instruction to do it efficiently
  - But, need to deal with the overflow problem
  - When the process terminates, the PCB vanishes
- Save the PCB inside Kernel
  - May not be as efficient as saving it on stack
  - But, it is very flexible and no other problems

Manipulating Processes

- Creation and termination
  - `fork, exec, wait, kill`
- Interaction
  - message passing between processes
- Syscalls include
  - `block, yield`

Threads

- thread
  - a sequential execution stream within a process (sometimes called a lightweight process)
  - threads in a process share the same address space
- thread concurrency
  - easy to program overlapping of computation with I/O
  - supports doing many things at a time: web browser
  - a server serves multiple requests

Thread Control Block (TCB)

- state (ready, running, blocked)
- registers
- status (EFLAGS)
- program counter (EIP)
- stack
- code

Thread API

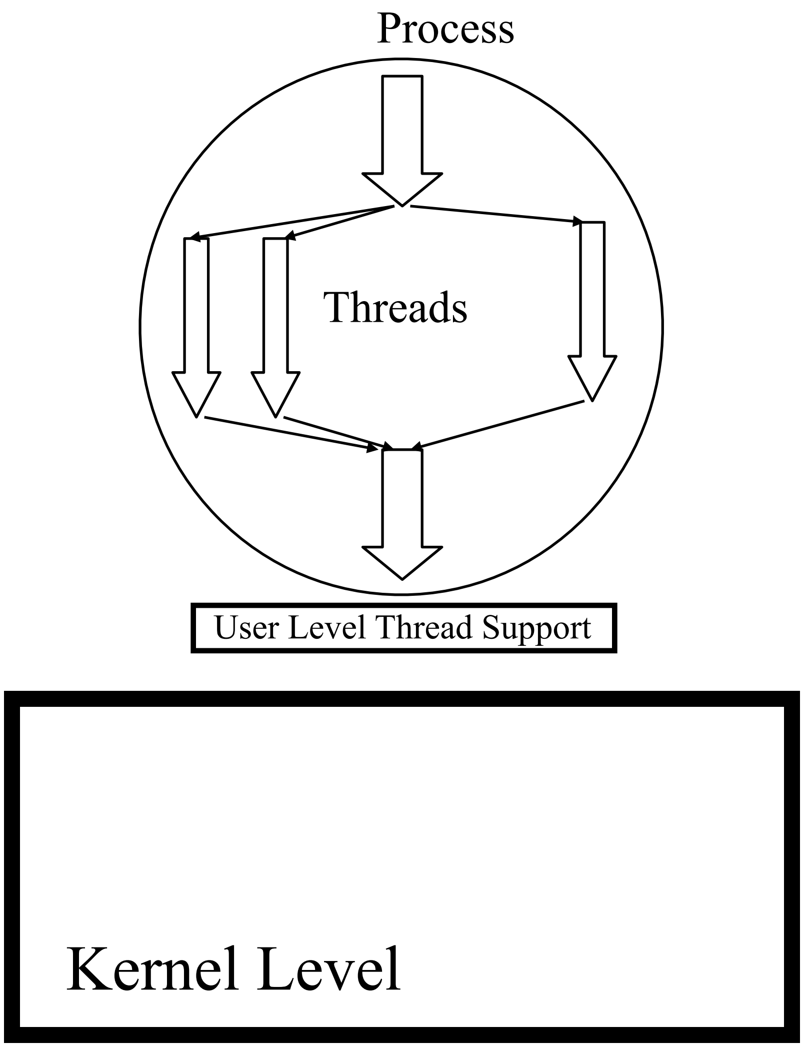
- creation
  - fork, join
- interaction
  - condition synchronization & mutual exclusion
    - acquire(lock_name), release (lock_name)
    - semaphores
  - operations on monitor *condition variables*
    - wait, signal, broadcast

Process vs. Thread

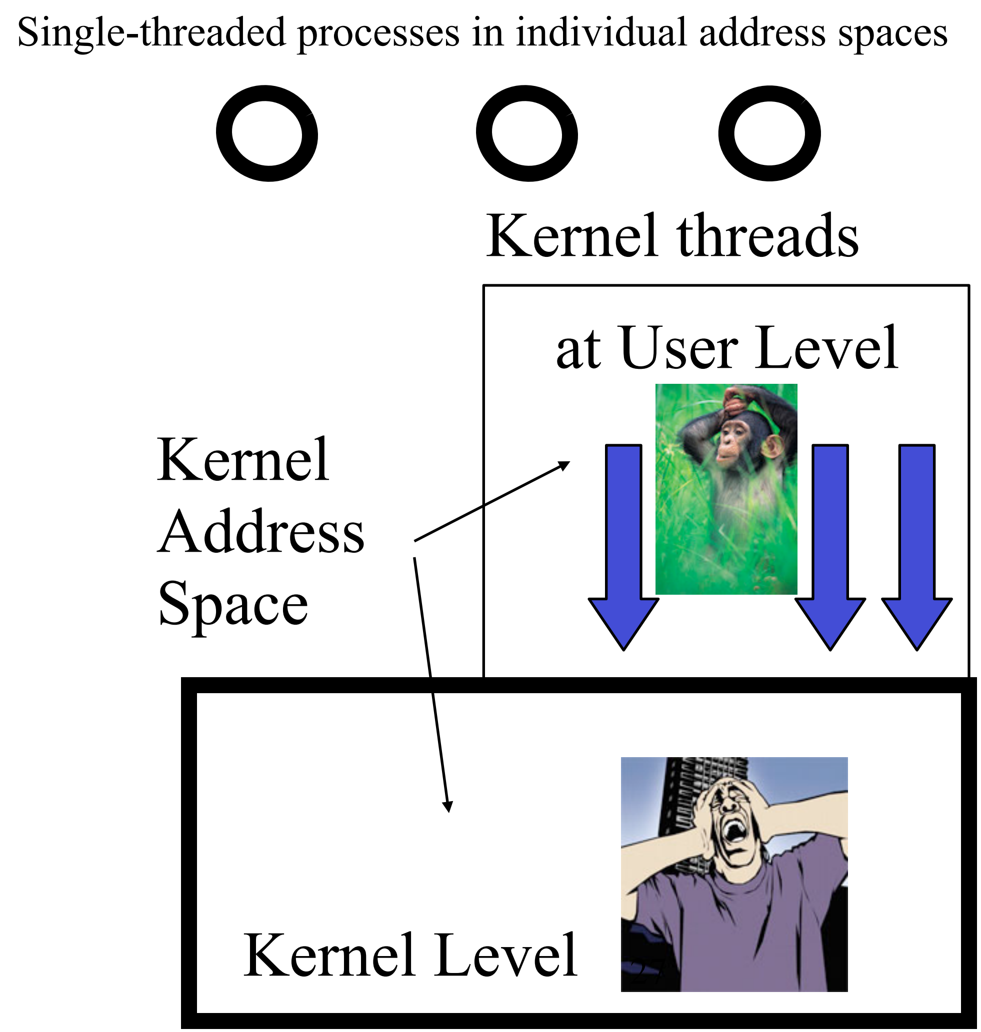
- address space
  - processes do not (usually) share memory, threads in a process do
    - therefore, process context switch implies getting a new address space in place
      - page table and other memory mechanisms
- privileges
  - each process has its own set, threads in a process share

Threads and Processes in this Course

Trad. User-Level Threads



Project OS



User- and Kernel-Level Thread Support

- **User-level** threads within a process are
  - Not seen by Kernel (so Kernel can not block and schedule them)
  - Scheduled by (user-level) scheduler in process
- **Kernel-level** threads
  - Seen by OS Kernel (so Kernel can block and schedule them individually)

User- and Kernel-Level Thread Support

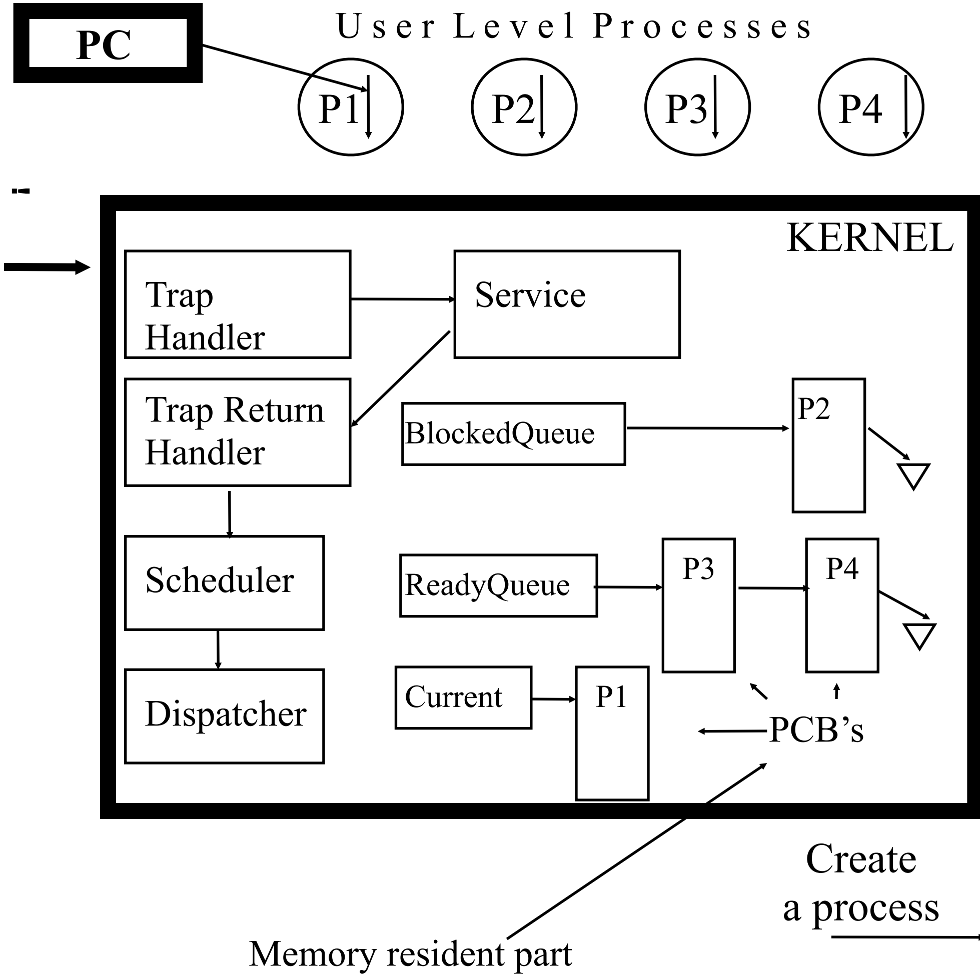
- **User-level** threads within a process are
  - Not seen by Kernel (so Kernel can not block and schedule them)
  - Scheduled by (user-level) scheduler in process
- **Kernel-level** threads
  - Seen by OS Kernel (so Kernel can block and schedule them individually)

What if a thread blocks?

Context Switching Issues

- Performance
  - Overhead multiplied so need to keep it fast (nano vs micro vs milli seconds)
  - Most time is spent SAVING and RESTORING the context of processes
    - Less processor state to save, the better
      - Pentium has a multitasking mechanism, but SW can be faster if it saves less of the state
    - How to save time on the copying of context state?
      - Re-map (address) instead of **copy** (data)
- Where to store Kernel data structures "shared" by all processes
    - Memory
- How to give processes a fair share of CPU time
    - Preemptive scheduling, time-slice defines maximum time interval between scheduling decisions

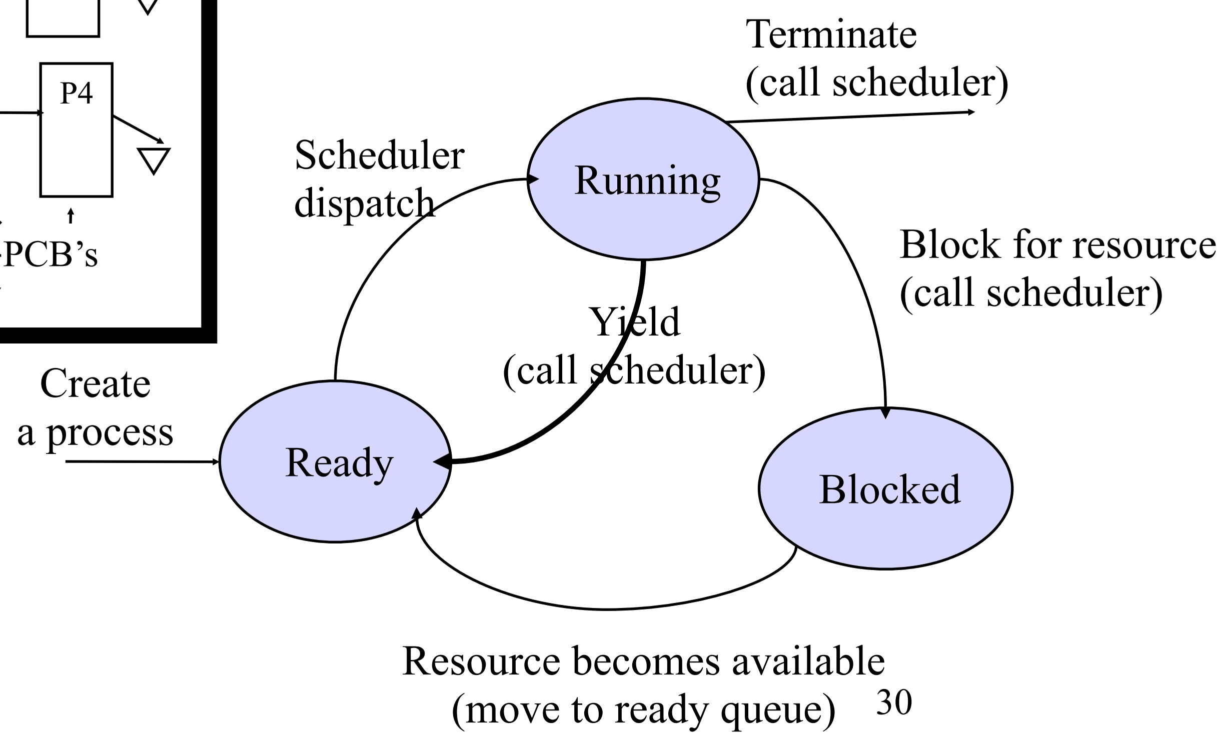
Example Process State Transitions



MULTIPROGRAMMING

•Uniprocessor: *Interleaving* ("pseudoparallelism")

•Multiprocessor: *Overlapping* ("true paralellism")



Scheduler

- Non-preemptive scheduler invoked by **syscalls** (to OS Kernel)
  - block
  - yield
  - (fork and exit)
- The simplest form

Scheduler:

```
save current process state (store to PCB)
choose next process to run
dispatch (load state stored in PCB to registers, and run)
```

- Does this work?
  - **PCB must be resident in memory**
  - **Remember the stacks**

Process Context Switch

- save a context
  - all registers (general purpose ad floating-point)
  - all co-processor state
  - save all memory to disk?
  - what about cache and TLB?
- start a context: reverse of above
- challenge: save state without changing it before it is saved
  - hardware will save a few registers when an interrupt happens. We can use them.
  - CISC: have a special instruction to save and restore all registers to/from stack
  - RISC: reserve registers for kernel

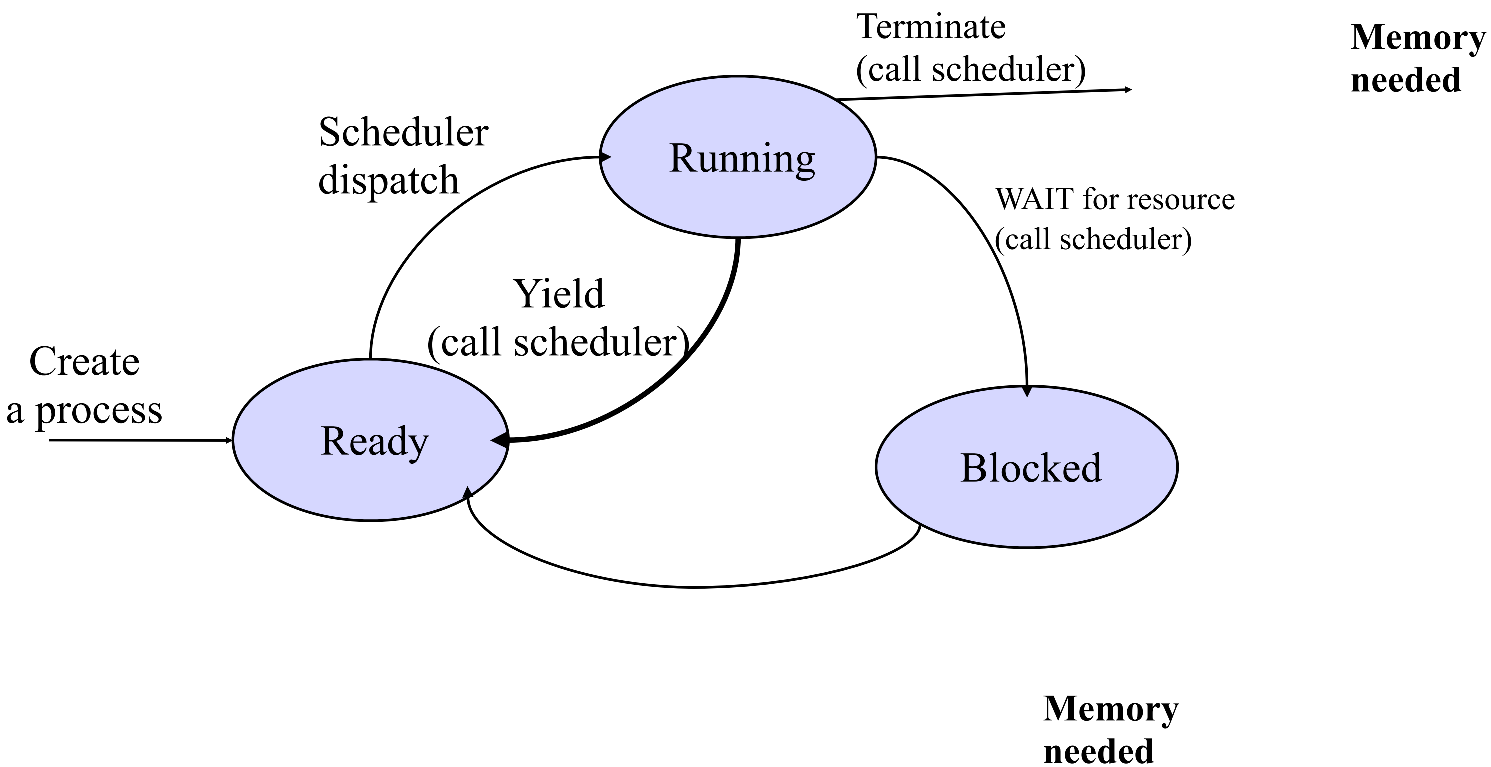
Stacks

- Remember: *We have only one copy of the Kernel in memory*
  - Here is a way to view this: all processes "execute" the same kernel code (=> Must have a kernel stack for each process)
- Used for storing parameters, return address, locally created variables in *frames* or *activation records*
- Each process
  - user stack
  - kernel stack
    - always empty when process is in user mode executing instructions
- Does the Kernel need its own stack(s)?

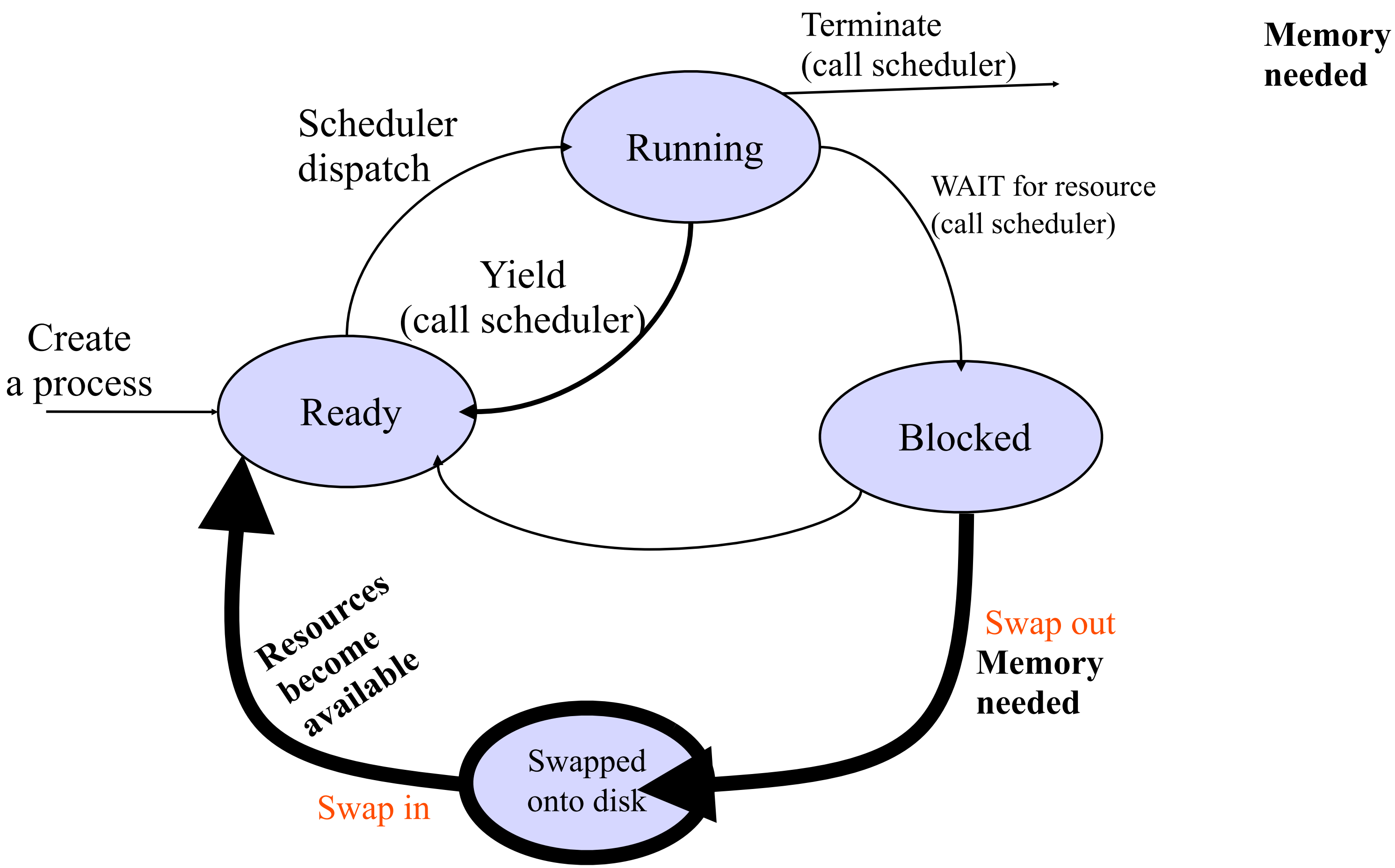
"Swapping"

- The processes competing for resources may have combined demands that exceeds available resources (like memory)
- Reducing the degree of multiprogramming by moving some processes to disk, and temporarily not consider them for execution may be a strategy to provide for "infinite pie"

Add Job Swapping to State Transition Diagram



Add Job Swapping to State Transition Diagram



Add Job Swapping to State Transition Diagram