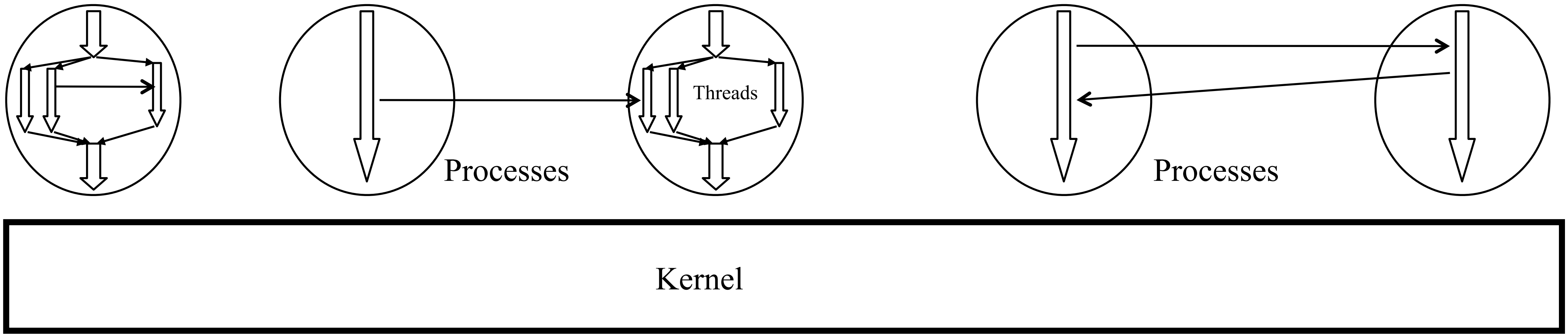
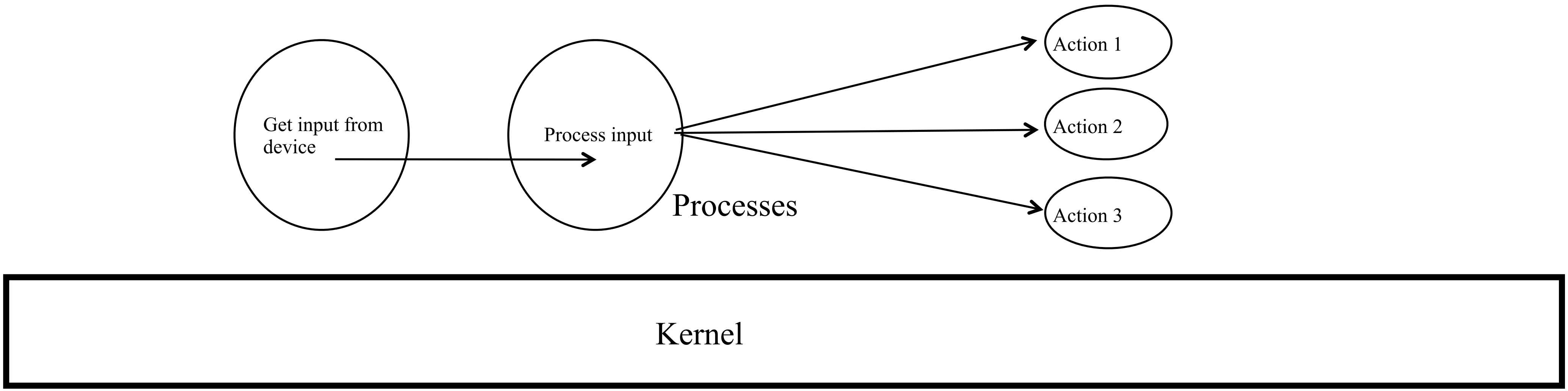
Processes and Non-Preemptive Scheduling

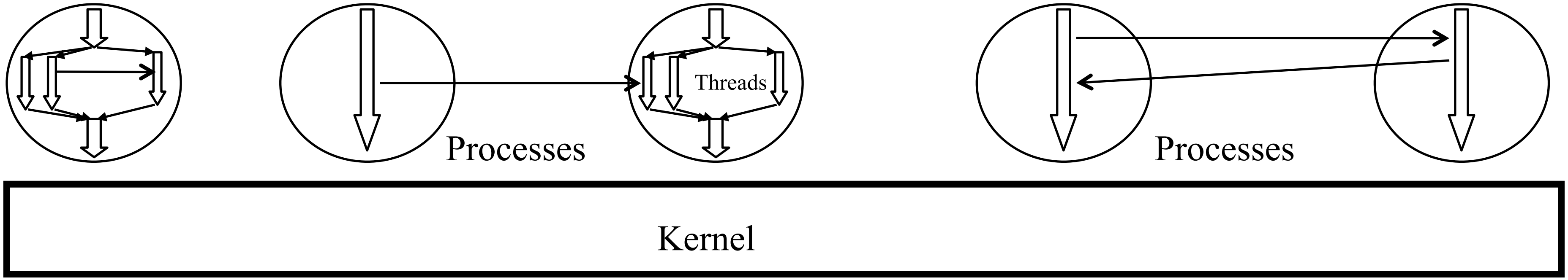
Otto J. Anshus

Concurrency and Process

- Challenge: Physical reality is Concurrent
  - Smart to do “concurrent software” instead of sequential?
  - At least 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 and faults (page faults) will happen
  - Options
    - let each application/computation see the others and deal with it (each must fight or cooperate with the others)
    - let each application/computation believe it has the computer all alone (analogy: each car sees the highway without other cars (but perhaps it is a highway where the width and the speed limit can change at any time)
- Trad. approach:
  - Make the OS understand “process” and support processes
  - 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 at a time
    - Each process can believe it has a computer to itself: it can be written as if this is indeed the case


Threads
Processes
Processes
Kernel

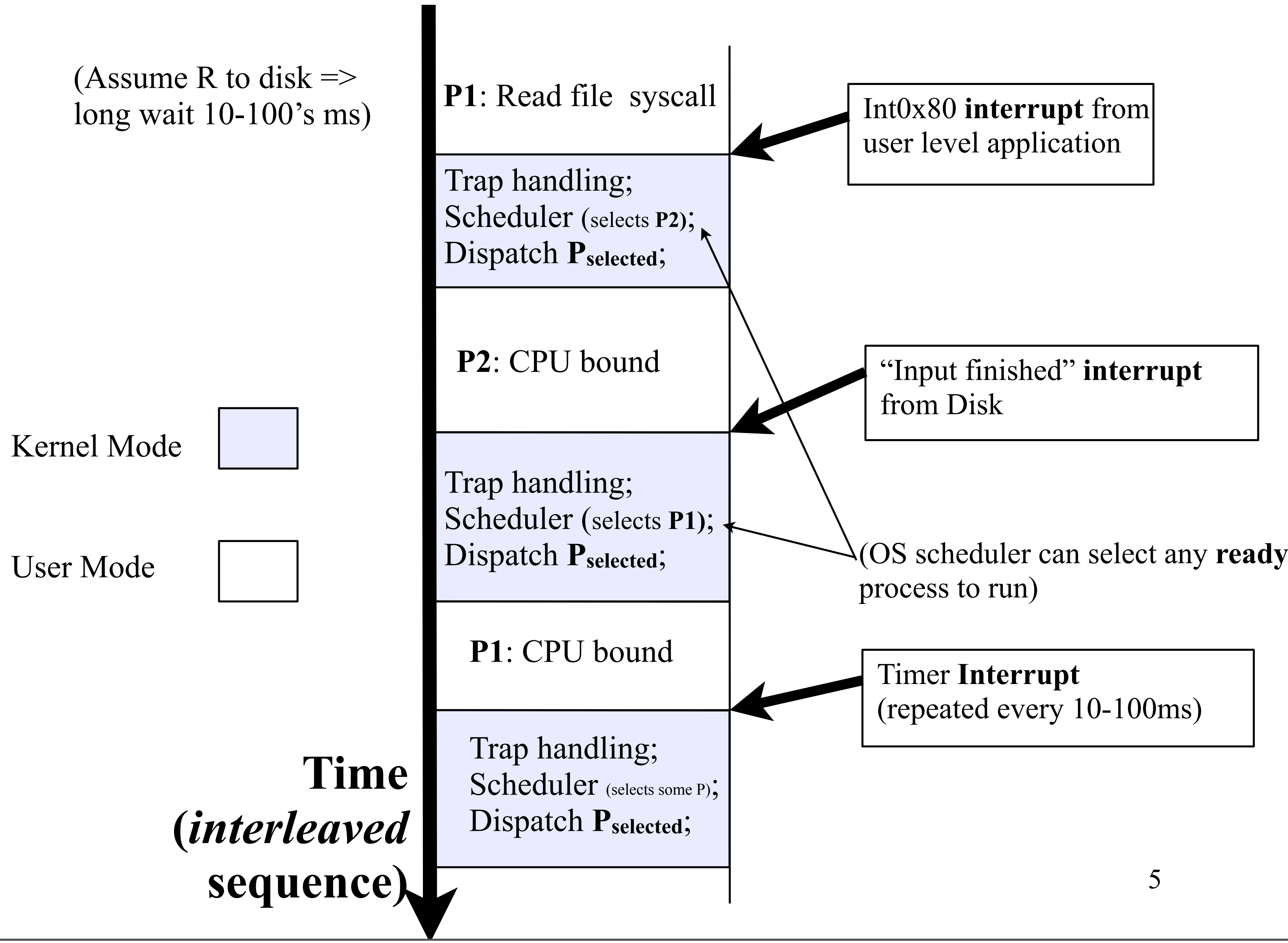

Get input from device
Process input
Processes
Action 1
Action 2
Action 3
Kernel



Process

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

Flow of Execution



5

Concurrency & performance

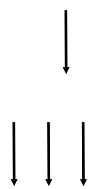
- Common in physical reality
- Speedup
  - ideal: n processes, n speedup
  - reality: bottlenecks + overheads
    - + sequential vs. parallel parts if & when the processes cooperate
  - Questions
    - Speedup when
      - working with 1 partner?
      - working with 20 partners?
    - Super-linear speedup?
  - Also check out Amdahl's Law

Procedure, Co-routine, Thread, Process

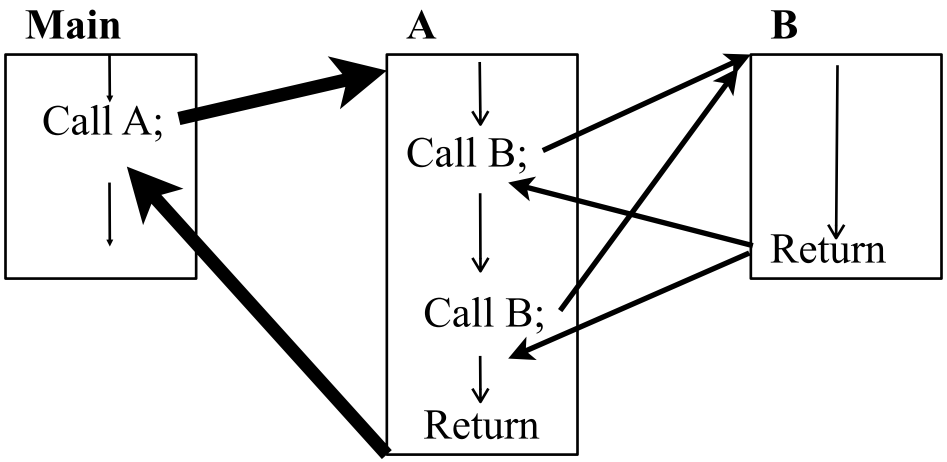
- Procedure, Function, (Sub)Routine
    - Call-execute all-return nesting
- Co-routine
    - Call-resumes-return

User level non preemptive “scheduler” in user code

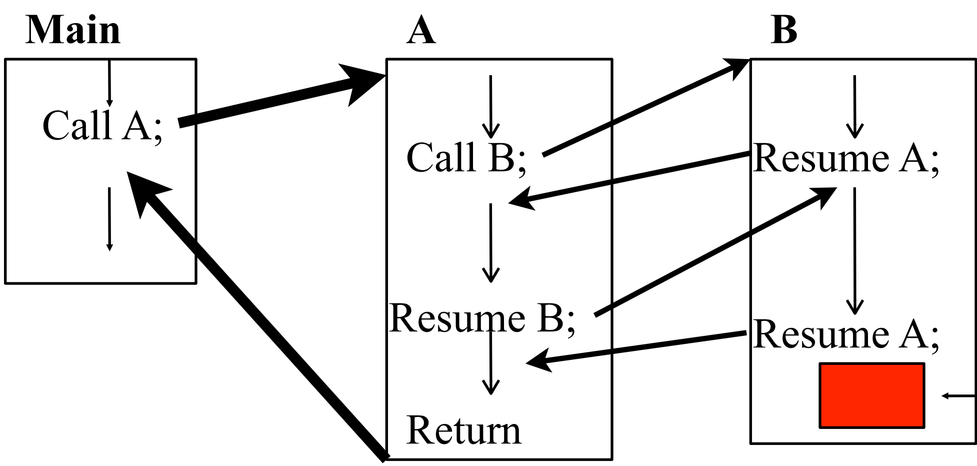
- Thread (more later)
- Process
    - Single threaded
    - Multi threaded



Procedure and Co-routine



"User Yield when finished"



"User 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 have at least one thread
  - Every thread exists within the context of a process?

Simplest (single threaded, sequential) Process

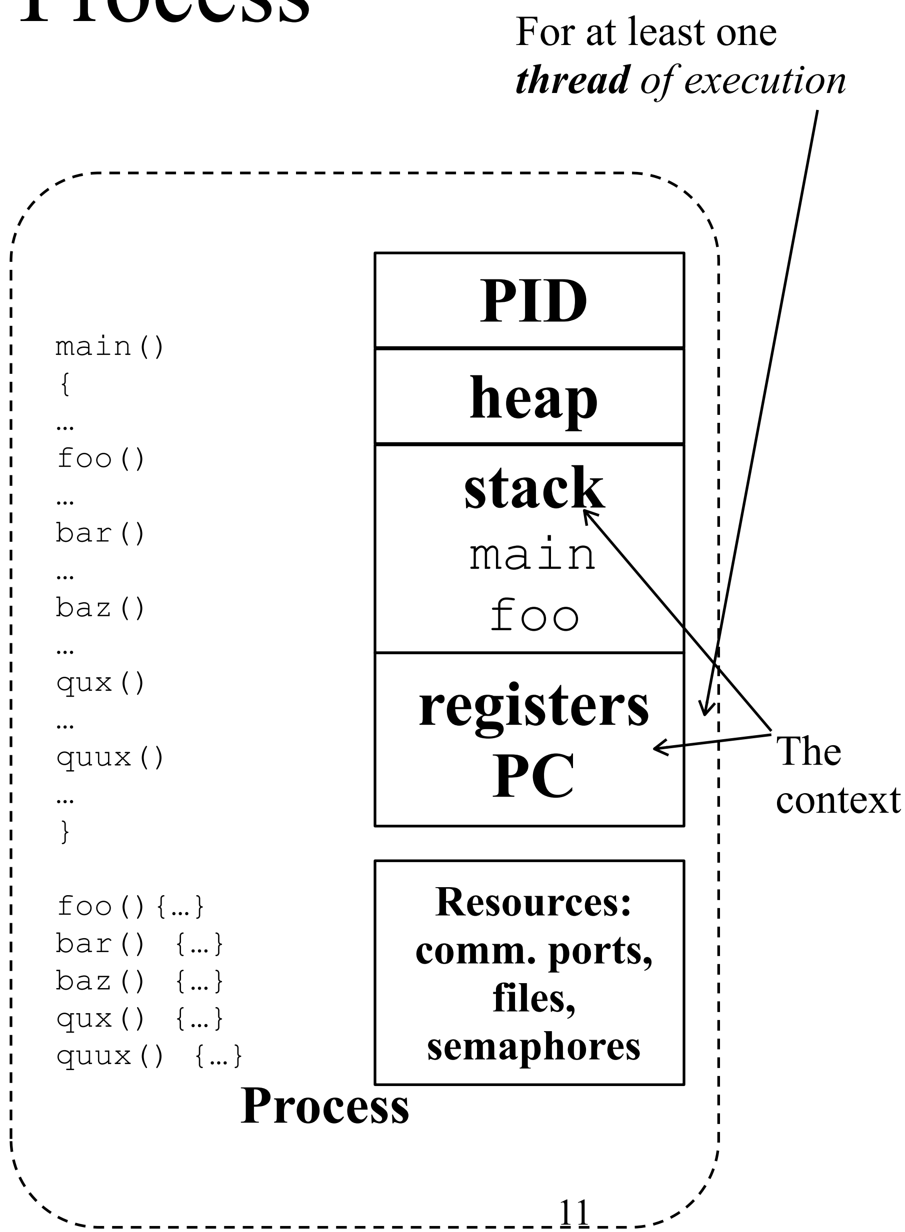
- Sequential execution of operations
  - No concurrency inside a (**single** threaded) process
  - Everything happens sequentially
- Process state
  - Registers
  - Stack(s)
  - Main memory
  - I/O devices
    - Files and their state
    - Communication ports
  - Other resources

Program and Process

```
main()
{
…
foo()
…
bar()
…
baz()
…
qux()
…
quux()
…
}

foo(){…}
bar() {…}/* a.k.a. gazonk
baz() {…}
qux() {…}
quux() {…}
```

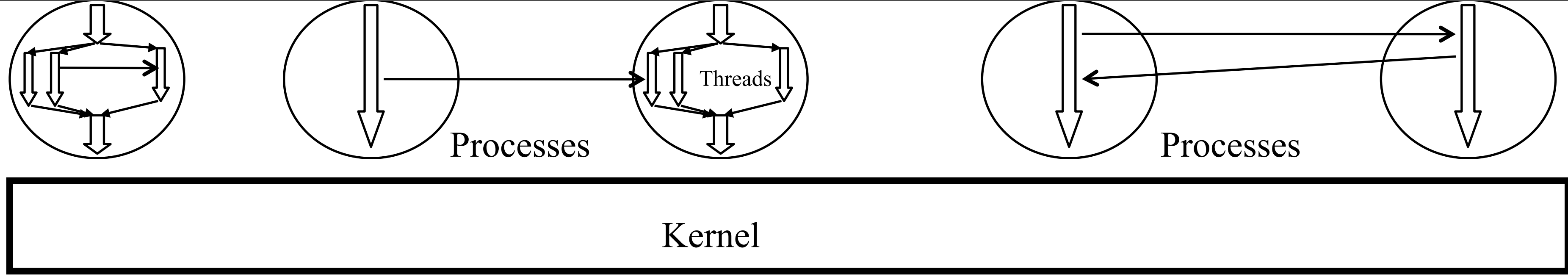
Program



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

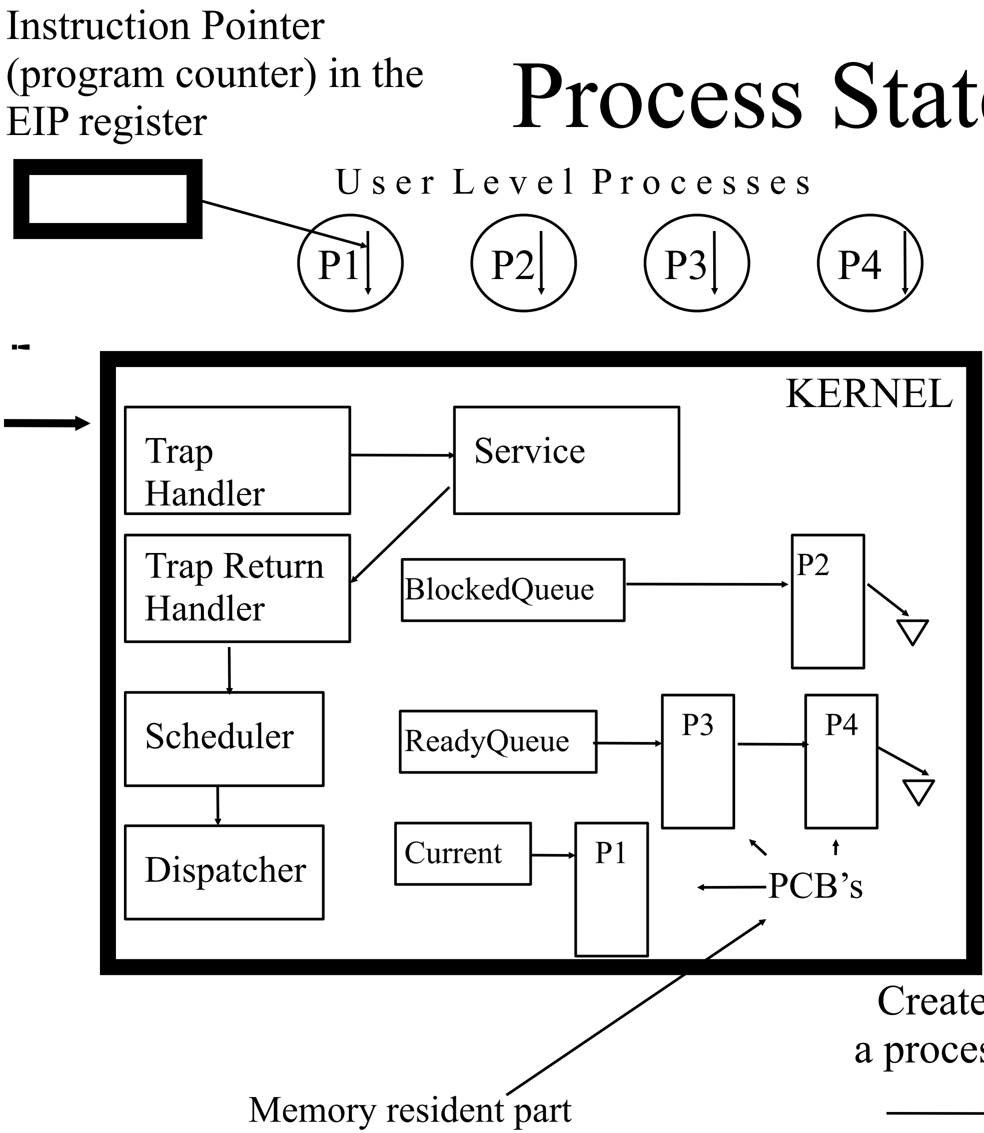
12



Supporting and Using Processes

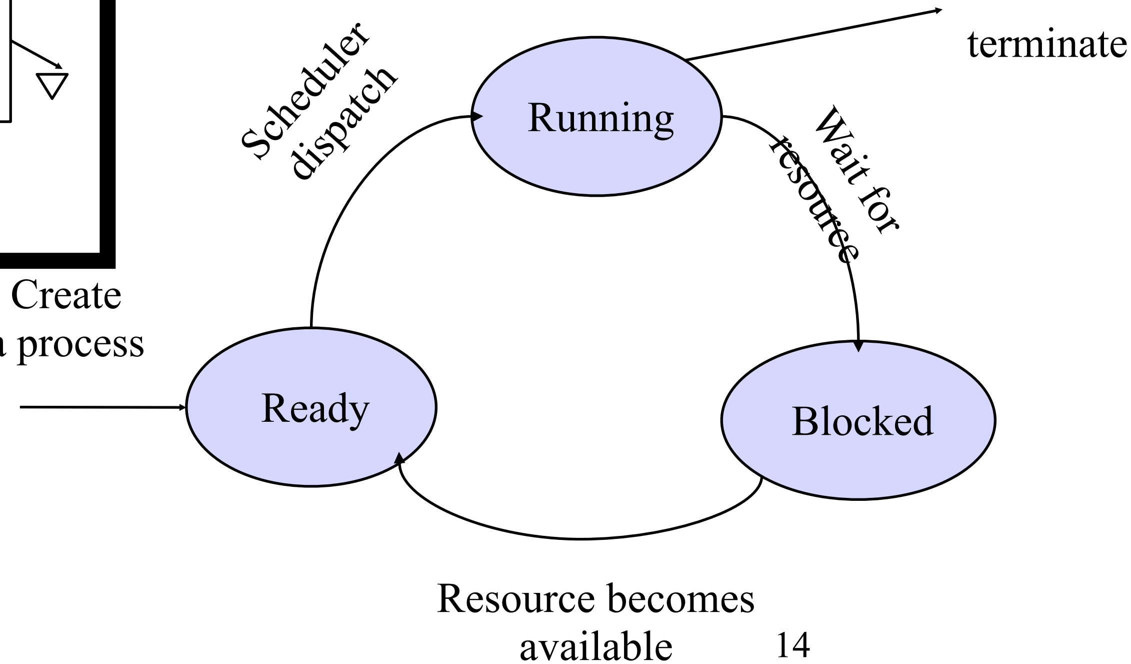
- Multiprogramming
  - Supporting concurrent execution *(overlapping or (transparently) interleaved)* of multiple processes (or multiple threads if only one process per program.)
  - Achieved by process- or context switching, switching the CPU(s) back and forth among the individual processes (threads), keeping track of each process' (threads) progress
- Concurrent programs
  - Programs that exploit multiprogramming for some purpose (e.g. performance, structure)
  - Independent or cooperating
  - Operating systems is important application area for concurrent programming. Many others (event driven programs, servers, ++)

Process State Transitions



MULTIPROGRAMMING

- Uniprocessor: *Interleaving* ("pseudoparallelism")
- Multiprocessor: *Overlapping* ("true paralellism")



14

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

- **Save**(volatile machine state, current process);
- **Load(**another process's saved volatile state);
- **Start**(new process);

16

Implementing processes

- OS (kernel) needs to keep track of all processes
  - Keep track of it's progress
  - (Parent process, if such a concept has been added)
  - Metadata (priorities etc.) used by OS
  - Memory management
  - File management
- 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*

Primitives of Processes

- Creation and termination
  - `fork, exec, wait, kill`
- Signals
  - Action, Return, Handler
- Operations
  - `block, yield`
- Synchronization
  - We will talk about this later

Processes (II)

- Classical/traditional processes were, using today's terminology, **Single Threaded**
- Sequential program
  - Single process
- Parallel program
  - Multiple cooperating processes

Threads

- thread
  - a sequential execution stream within a process (a.k.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

22

Thread Control Block (TCB)

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

Thread API

- creation
  - fork, join
- mutual exclusion
  - acquire(lock_name), release (lock_name)
- operations on monitor *condition variables*
  - wait, signal, broadcast
- alert
  - alert, alertwait, testalert

Thread vs. Procedure

- threads may resume out of order
  - cannot use LIFO stack to save state
  - each thread has its own stack
- threads can be asynchronous
  - procedure is synchronous: can use compiler to save state, and restore
- multiple overlapping threads
  - multiple CPUs

25

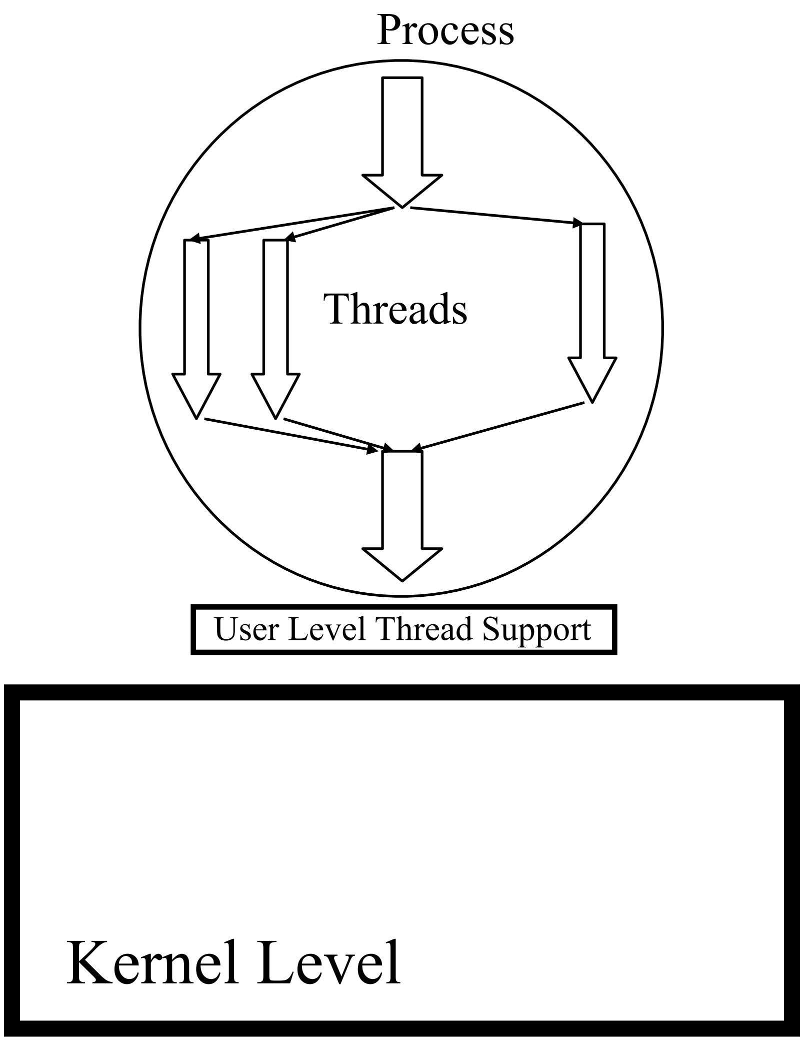
Process vs. Thread

- 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

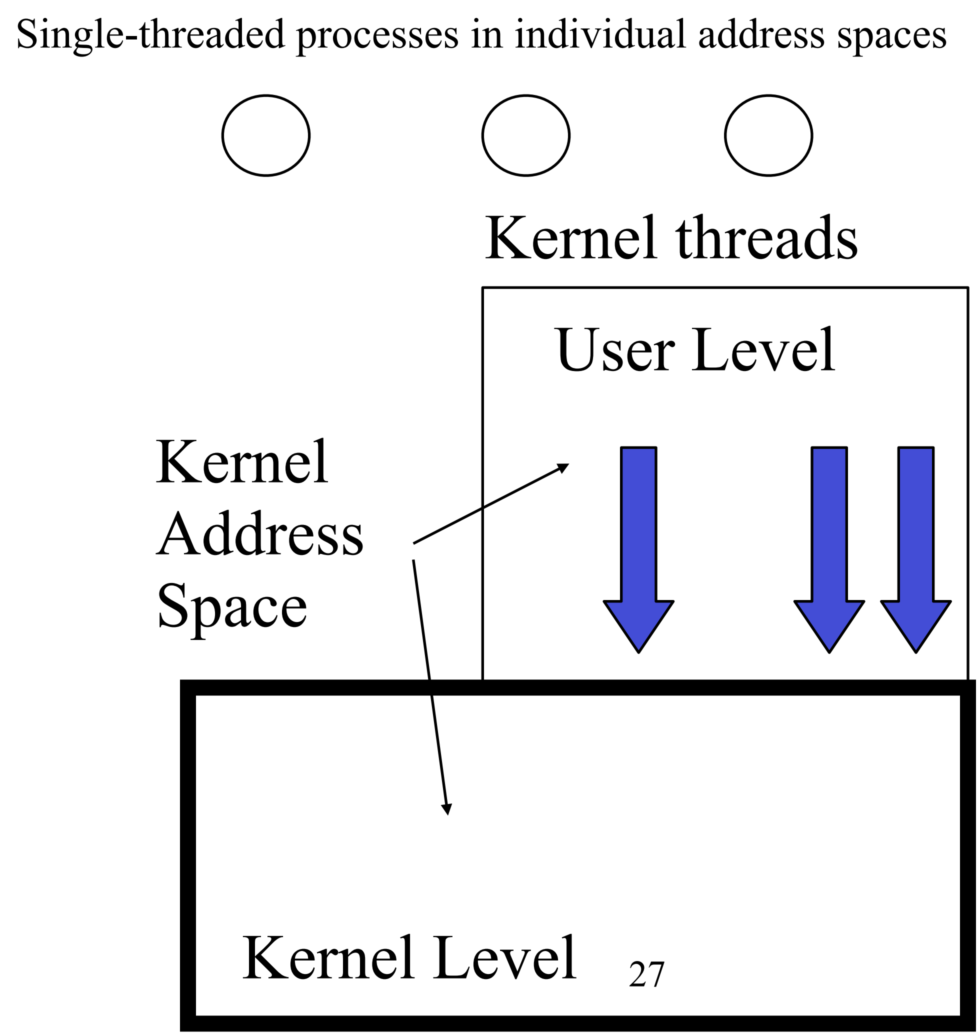
26

Threads and Processes in the Course Project OS

Trad. Threads



Project OS



User- and Kernel-Level Thread Support

- User-level threads within a process are
  - Indiscernible by OS
  - Scheduled by (user-level) scheduler in process
- Kernel-level threads
  - Maintained by OS
  - Scheduled by OS

28

User vs. Kernel-level Threads

- Question
  - What is the difference between user-level and kernel-level threads?
- Discussion
  - User-level threads are scheduled by a scheduler in their process at user-level
    - Co-routines
    - Cooperative scheduling (explicit "yield" syscall, implicitly at any syscall (Warning: shared resources can result in race conditions and deadlocks))
    - Timer interrupt to get preemption (Warning: shared resources)
  - Kernel-level threads are scheduled by kernel scheduler
  - Implications
    - When a **user**-level thread is blocked on an I/O event, the **whole process** is blocked
    - A context switch of **kernel** threads is more expensive than for user threads
    - A smart scheduler (two-level) can avoid both drawbacks. But is more complex
      - Do we like complexity?

29

Threads & Stack

- **Private**: Each user thread has its own kernel stack
- **Shared**: All threads of a process share the same kernel stack

	Private kernel stack	Shared kernel stack
Memory usage	More	Less
System services	Concurrent access	Serial access
Multiprocessor	Yes	No
Complexity	More	Less

Example: `fork` (UNIX)

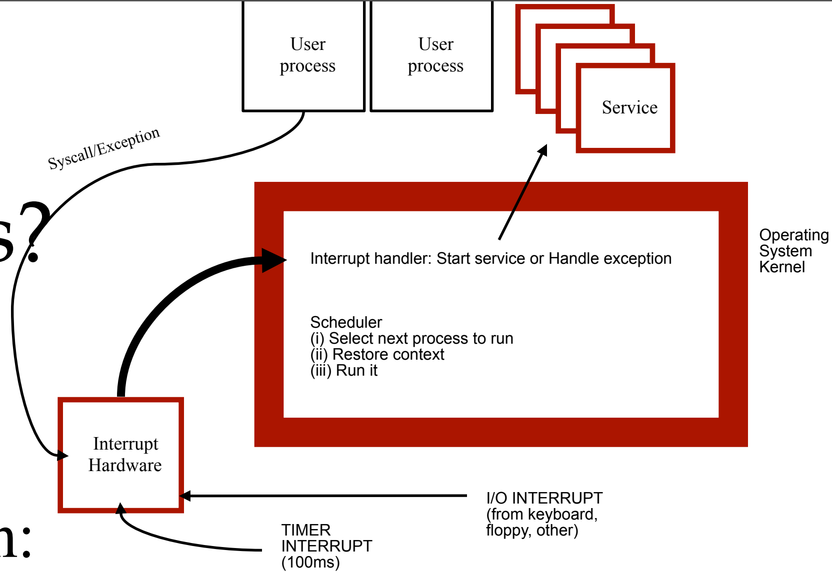
- fork() **clones** a process
  - Spawns a new process (with new PID)
  - Called in parent process
  - Returns in parent *and* child process
  - Return value in parent is child's PID
  - Return value in child is '0'
  - Child gets duplicate, but separate, copy of parent's user-level virtual address space
  - Child gets identical copy of parent's open file descriptors
- exec overlays (replaces) the current process
- ```
  if ((pid=fork())==0){
          /*child*/                exec("foo");          /*does not return*/}
  else    /*parent*/   wait(pid);  /*wait for child to terminate*/
  ```
  ```

fork, exec, wait, kill

- Return value tested for error, zero, or positive
- Zero, this is the child process
  - Typically redirect standard files, and
  - Call Exec to load a new program instead of the old
- Positive, this is the parent process
- Wait, parent waits for child's termination
  - Wait before corresponding exit, parent blocks until exit
  - Exit before corresponding wait, child becomes zombie (un-dead) until wait
- Kill, specified process terminates

32

When may OS switch contexts?



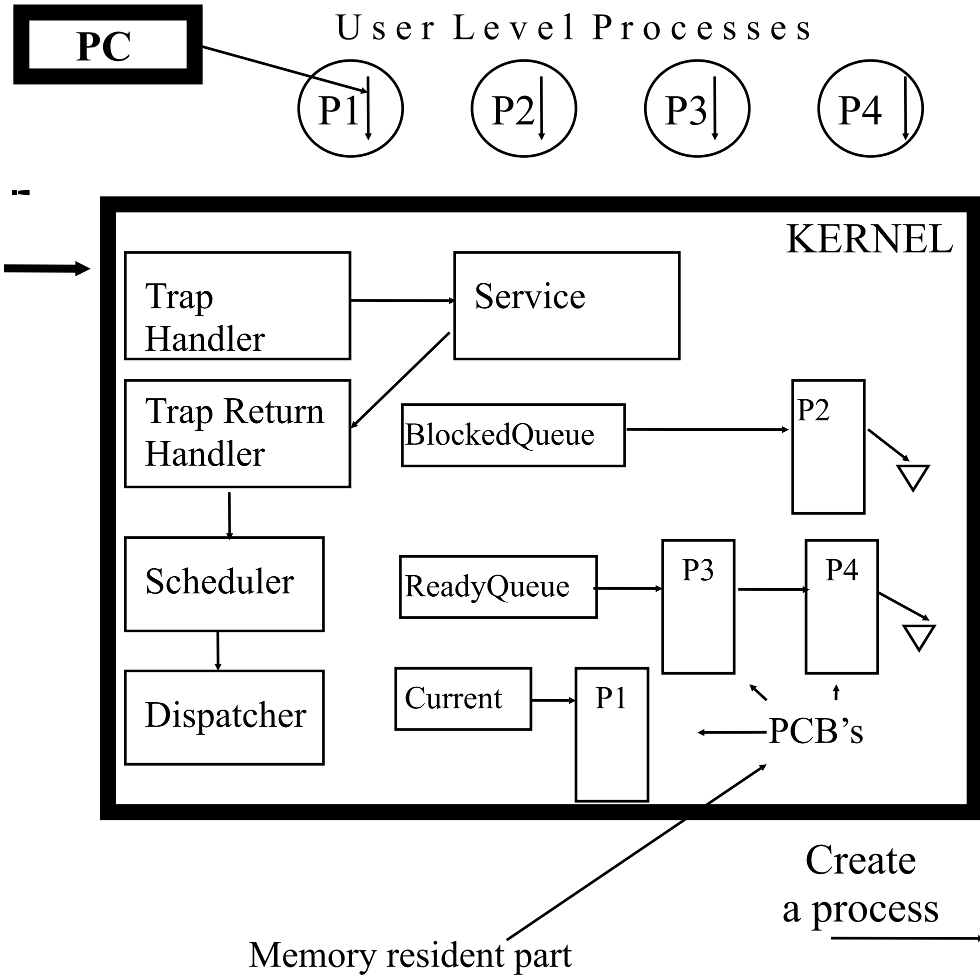
- Only when OS runs
- Events potentially causing a context switch:
  - (User level) system calls
    - Process created (`fork`)
    - Process exits (`exit`)
    - Process blocks implicitly (I/O calls, `block/wait`, IPC calls)
    - Process enters state **ready** explicitly (`yield`)
  - System Level Trap
    - By HW
    - By SW exception
  - Kernel preempts current process
    - Potential scheduling decision at "any of above"
    - *+"Timer" to be able to limit running time of processes*

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

34

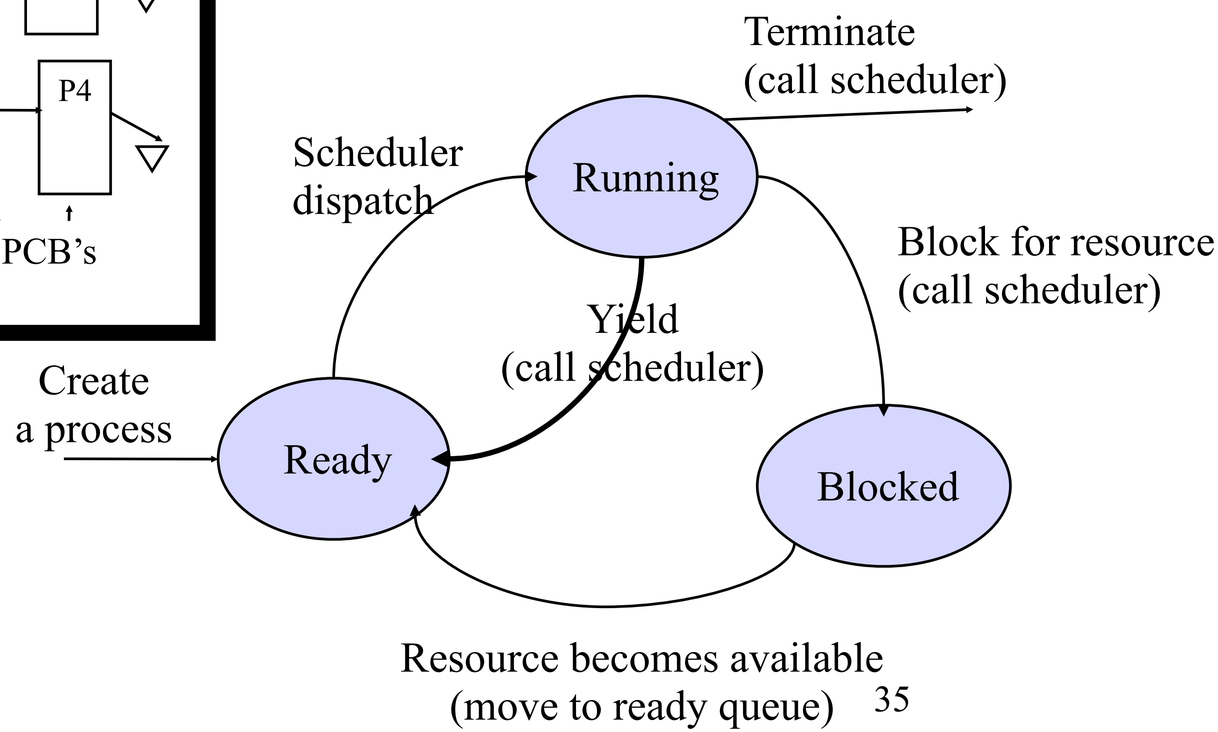
Example Process State Transitions



MULTIPROGRAMMING

•Uniprocessor: *Interleaving* ("pseudoparallelism")

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



35

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`**

Stacks

- Remember: *We have only one copy of the Kernel in memory*

  => all processes “execute” the same kernel code

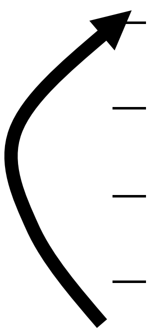
  => Must have a kernel stack for each process
- Used for storing parameters, return address, locally created variables in *frames* or *activation records*
- Each process
  - user stack
  - kernel stack
    - always empty when process is in user mode executing instructions
- Does the Kernel need its own stack(s)?

More on Scheduler

- Should the scheduler use a special stack?
  - Yes,
    - because a user process can overflow and it would require another stack to deal with stack overflow
    - (because it makes it simpler to pop and push to rebuild a process's context)
    - (Must have a stack when booting…)
- Should the scheduler simply be a "kernel process" (kernel thread)?
  - You can view it that way because it has a stack, code and its data structure
  - This thread always runs when there is no user process
    - "Idle" process
      - In kernel or at user level?

Win NT Idle

- No runable thread exists on the processor
  - Dispatch Idle Process (really a *thread*)
- Idle is really a dispatcher *in the kernel*
  - Enable interrupt; Receive pending interrupts; Disable interrupts;
  - Analyze interrupts; Dispatch a thread if so needed;
  - Check for deferred work; Dispatch thread if so needed;
  - Perform power management;



39

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

Where Should PCB Be Kept?

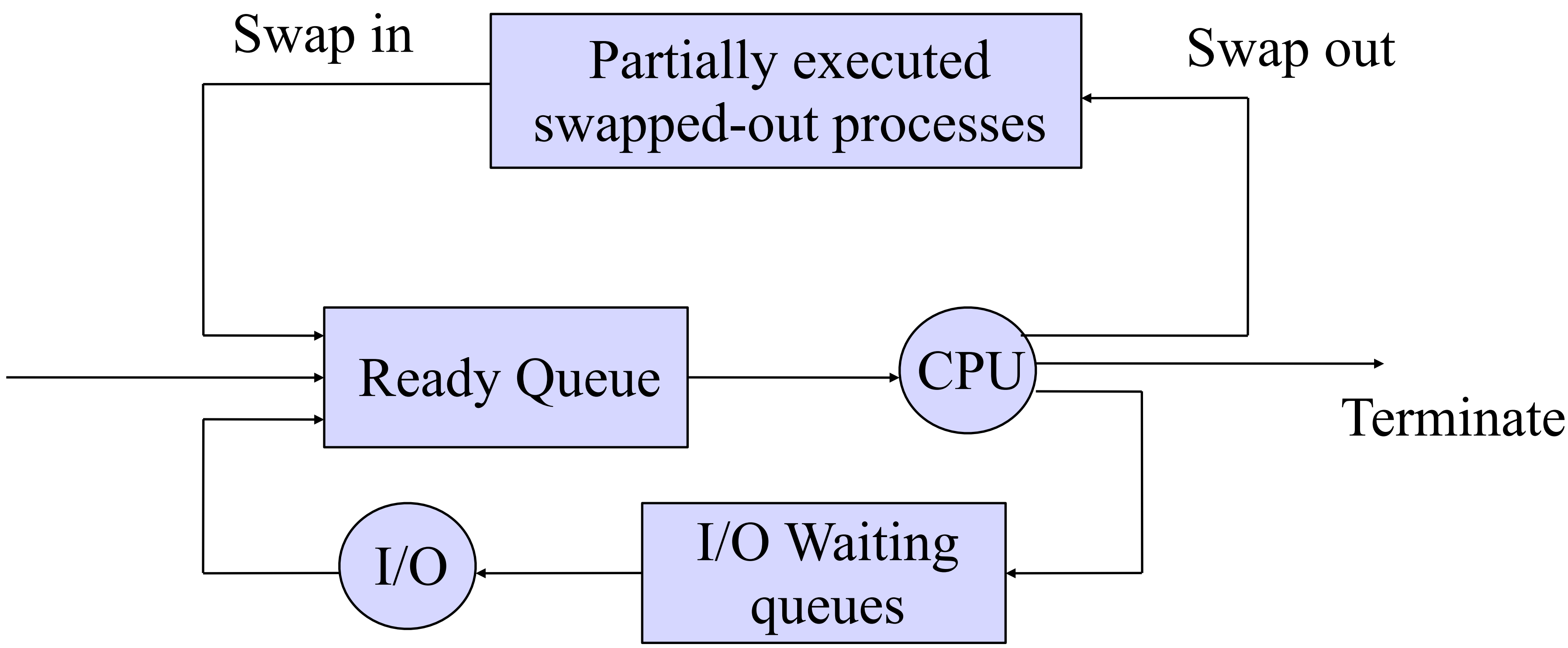
- Save the PCB on user stack
  - Many processors have a special instruction to do it efficiently
  - But, need to deal with the overflow problem
  - When the process terminates, the PCB vanishes
- Save the PCB on the kernel heap data structure
  - May not be as efficient as saving it on stack
  - But,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

Job swapping

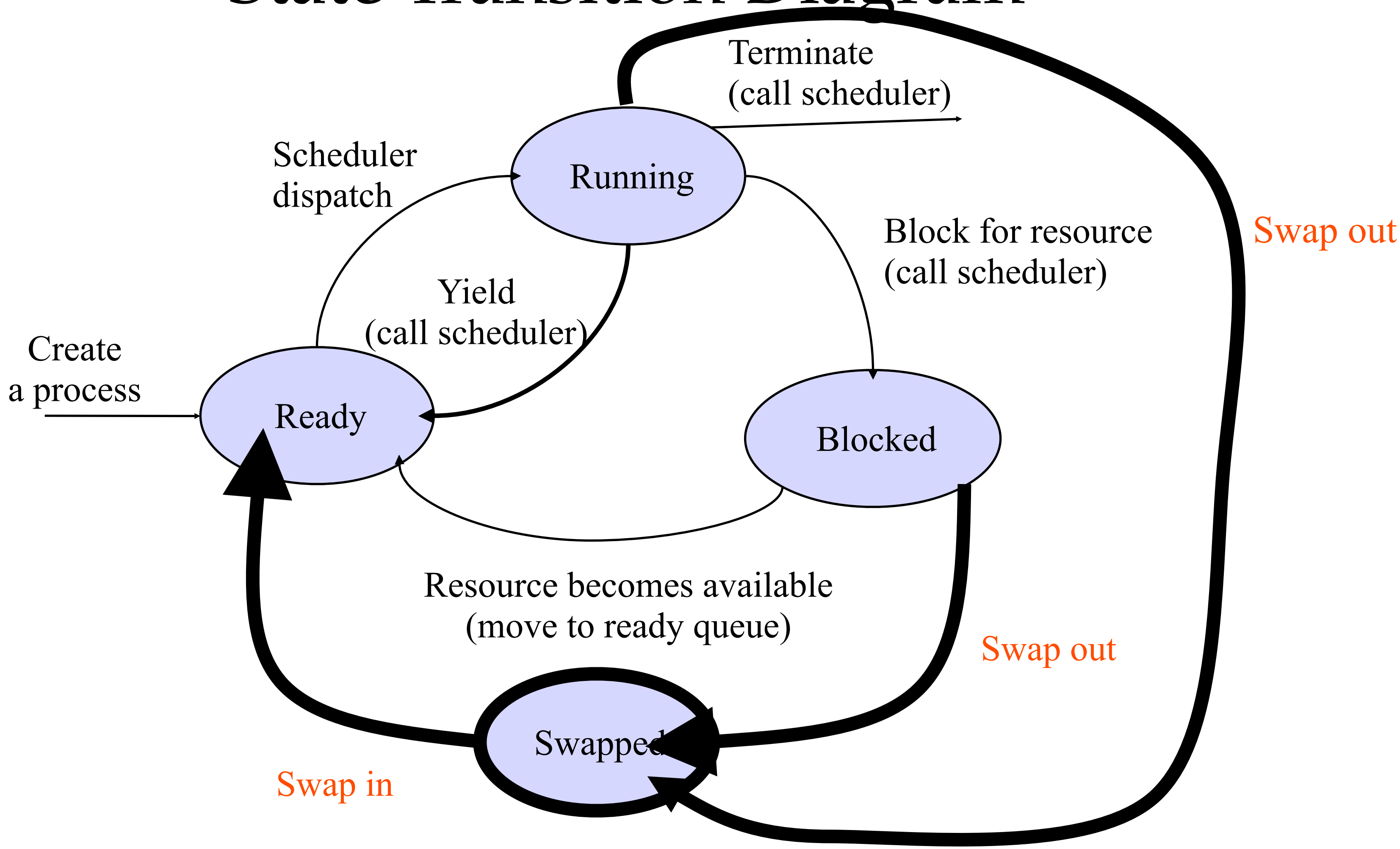
- The processes competing for resources may have combined demands that results in poor system performance
- Reducing the degree of multiprogramming by moving some processes to disk, and temporarily not consider them for execution may be a strategy to enhance overall system performance

42

Job Swapping



Add Job Swapping to State Transition Diagram



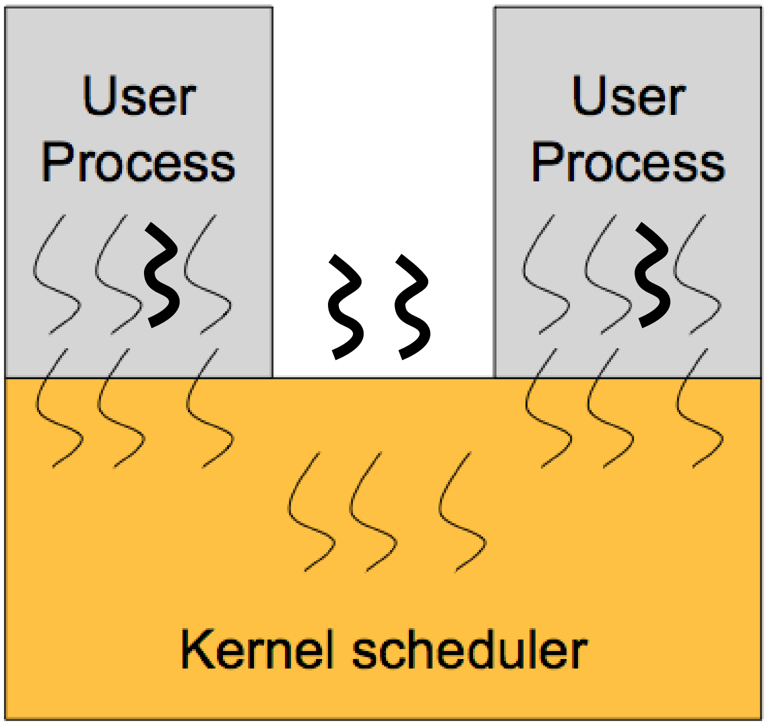
Concurrent Programming w/ Processes

- Clean programming model
  - User address space is private
    - Processes are protected from each other
    - Sharing requires some sort of IPC (InterProcess Communication)
- Overhead (slower execution)
  - Process switch, process control expensive
  - IPC expensive

Revisit Monolithic OS Structure

- All processes share the same kernel
- Kernel comprises
  - Interrupt handler & Scheduler
  - Key drivers
  - Threads "doing stuff"
  - Process & thread abstraction realization
  - Boot loader, BIOS
- Scheduler
  - Use a **ready queue** to hold all ready threads (=="process" if single-threaded)
  - Schedule a thread in
    - current
    - or a new context

We will have: Single threaded processes



We will have: Kernel with multiple threads (kind of)

46