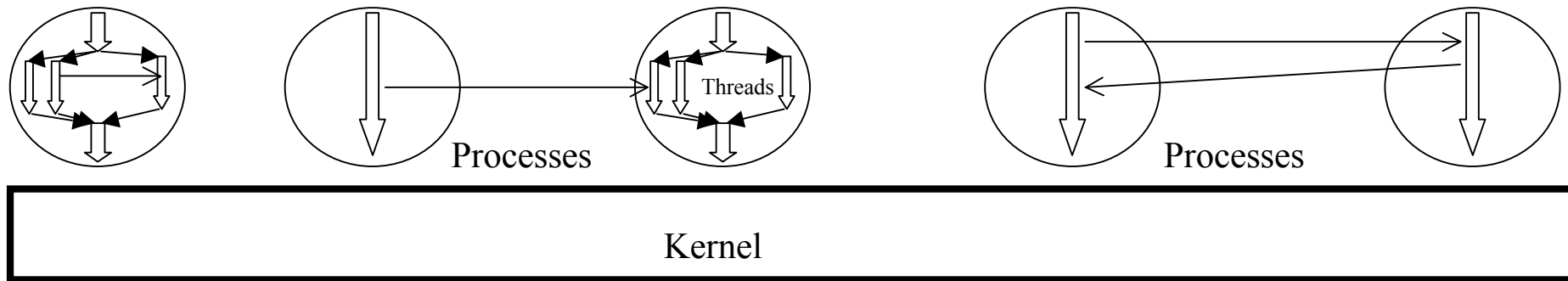


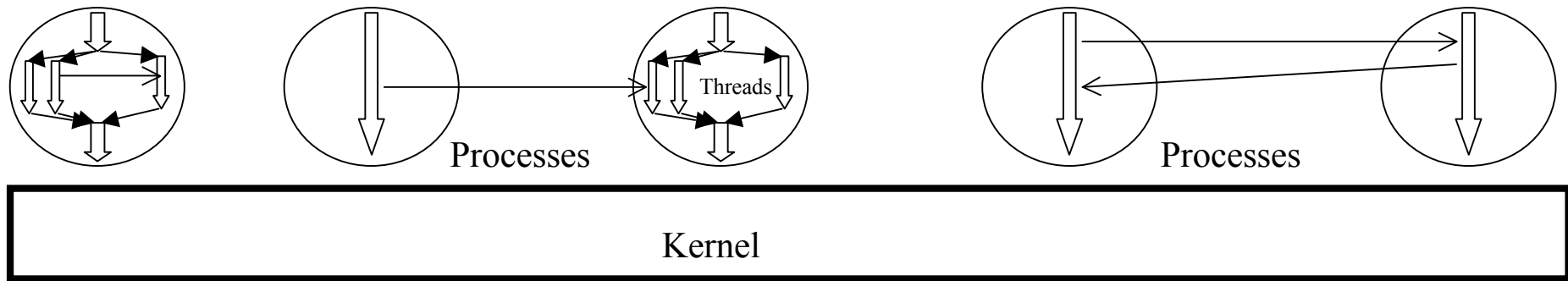
Processes and Non-Preemptive Scheduling

Otto J. Anshus



An aside on concurrency

- Timing and sequence of events are key concurrency issues
- We will study classical OS concurrency issues, including implementation and use of classic OS mechanisms to support concurrency
- A course on parallel programming may revisit this material
- In a course on distributed systems you may/may not want to use (formal?) tools to understand and model timing and sequencing better
- Single CPU computers are designed to uphold a simple and rigid model of sequencing and timing. "Under the hood," even single CPU systems are distributed in nature, and are carefully organized to uphold strict external requirements
- Emerging CPUs (2005) are set to be increasingly multi core, multi threaded dies across market segments.
 - *Challenges and opportunities to design cost-effective high-performance systems.*



Process

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

Processes (II)

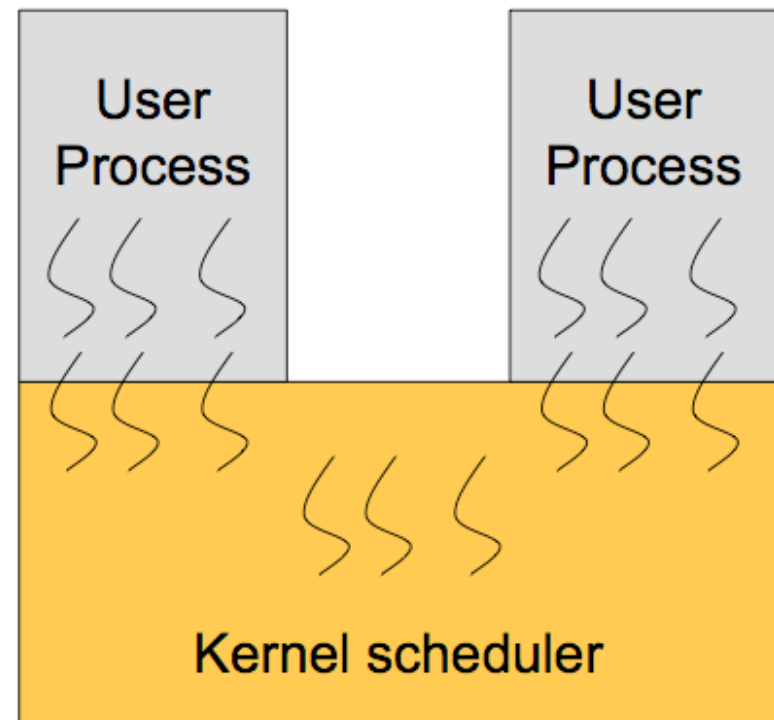
- Classical processes were, using today's terminology; “Single Threaded”
- Sequential program: Single process
- Parallel program: Multiple cooperating processes

Processes (III)

- “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 must contain one or more threads
- Every (?) thread exists within the context of a process

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



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

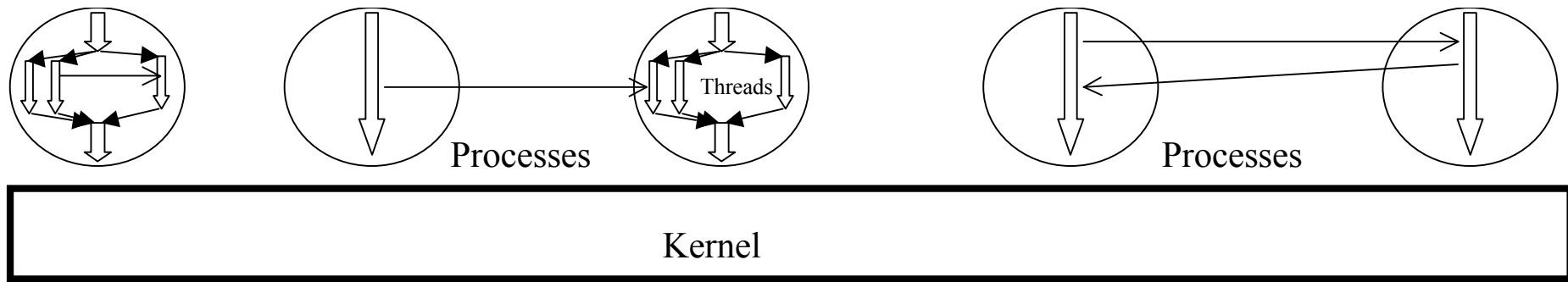
User vs. Kernel-level Threads

- Question
 - What is the difference between user-level and kernel-level threads?
- Discussions
 - User-level threads are scheduled by a scheduler in their process at user-level
 - Co-routines
 - Timer interrupt to get preemption
 - 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?

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



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, ++)

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 align processes in *queues*

Primitives of Processes

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

fork (UNIX)

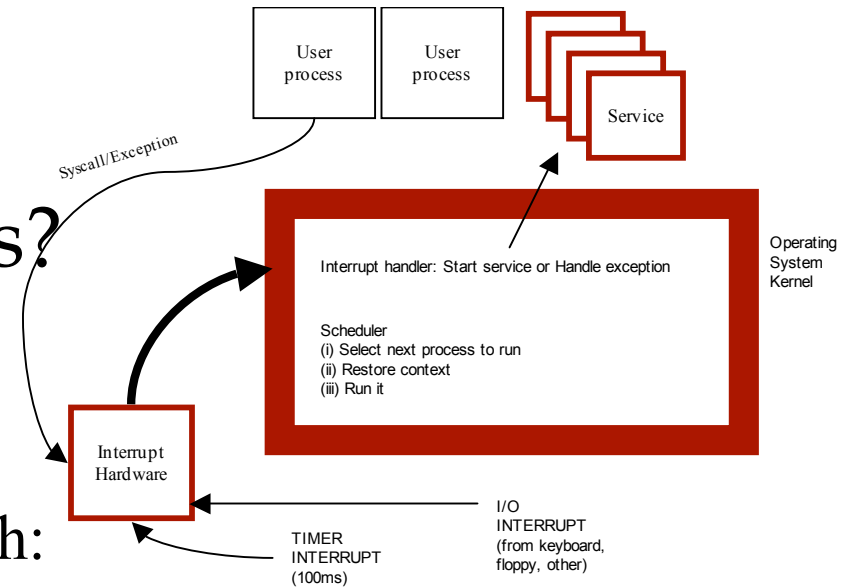
- Spawns a new process (with new PID)
- Called in parent process
- Returns in parent *and* child process
- Return value in parent is child's PID
- Return value in child is '0'
- Child gets duplicate, but separate, copy of parent's user-level virtual address space
- Child gets identical copy of parent's open file descriptors

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

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



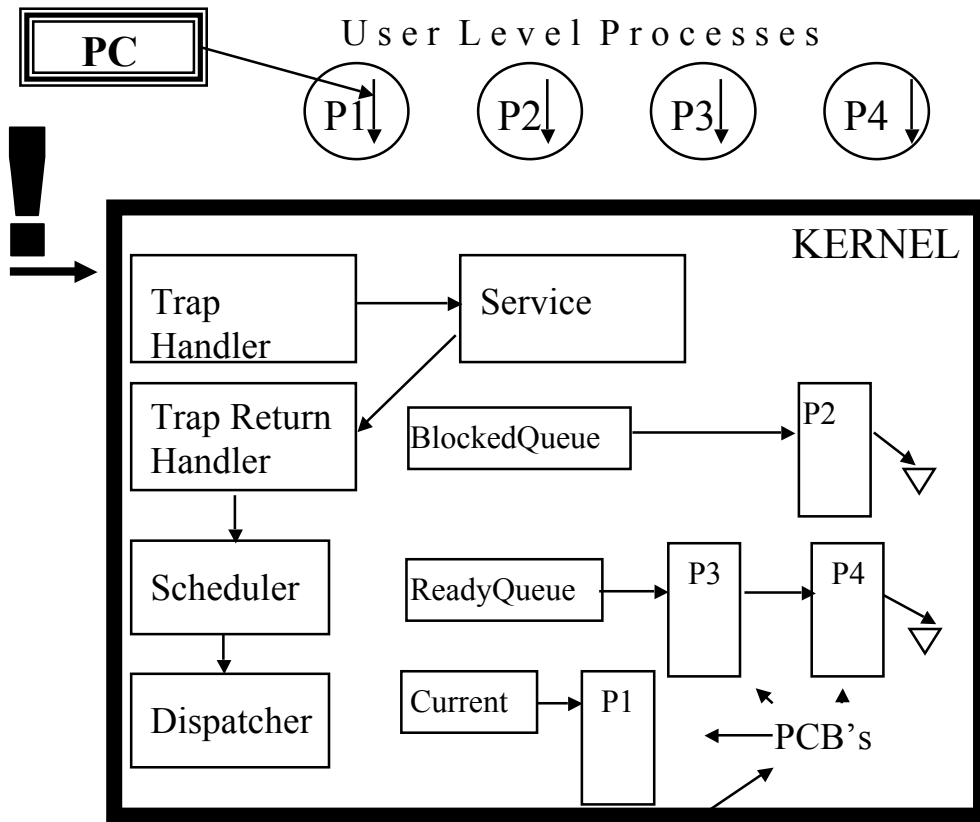
Non-Preemptive scheduling

Preemptive scheduling

Context Switching Issues

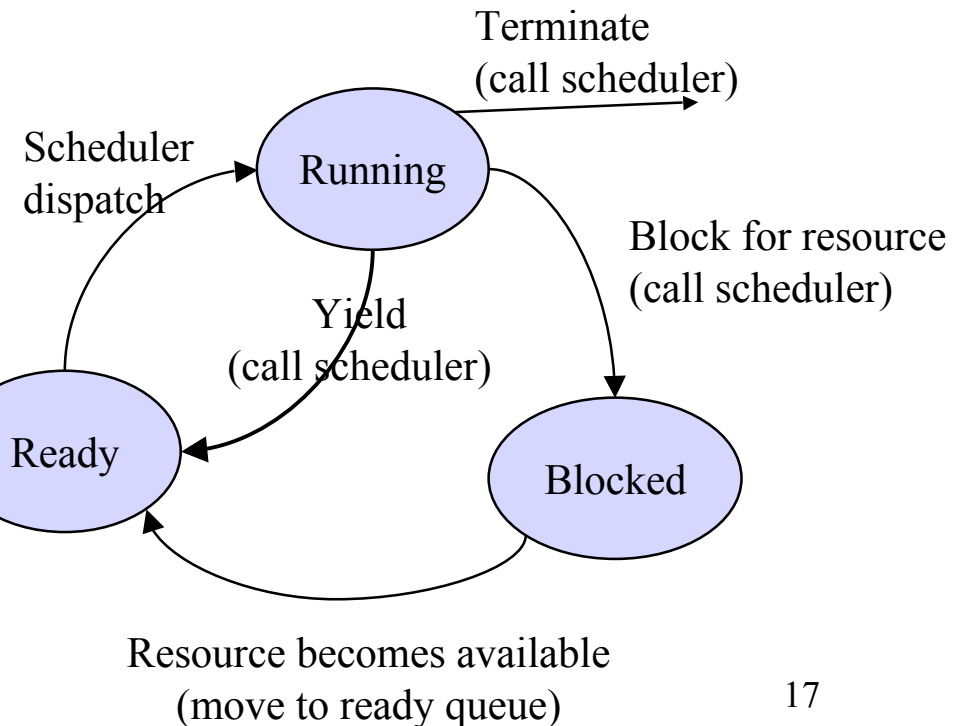
- Performance
 - Should be no more than a few microseconds
 - Most time is spent SAVING and RESTORING the context of processes
 - Less processor state to save, the better
 - Pentium has a multitasking mechanism, but SW can be faster if it saves less of the state
 - How to save time on the copying of 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



Memory resident part

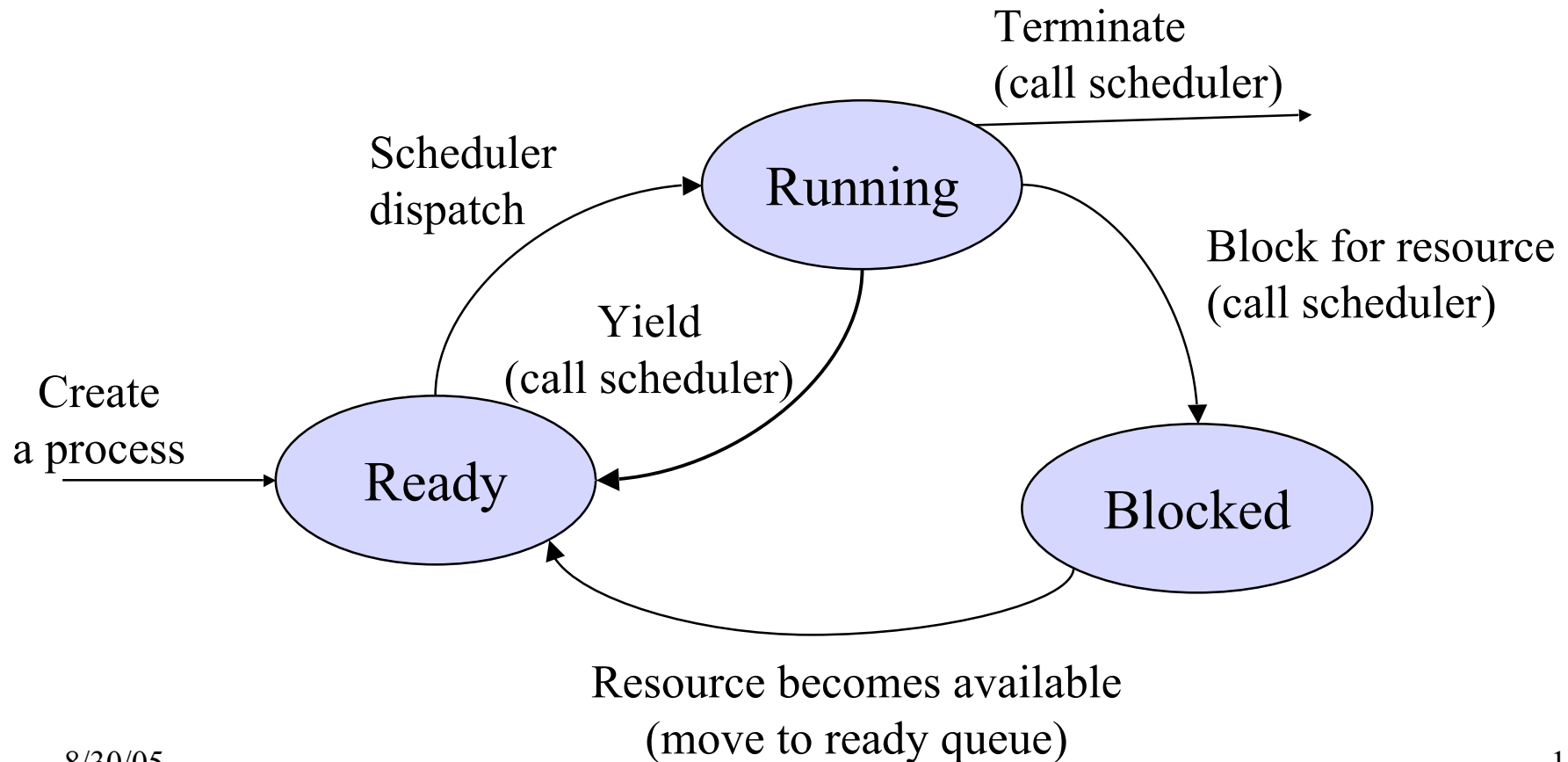
Create a process



MULTIPROGRAMMING

- Uniprocessor: *Interleaving* (“pseudoparallelism”)
- Multiprocessor: *Overlapping* (“true parallelism”)

Process State Transition of Non-Preemptive Scheduling



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 PCB and run)**
- Does this work?
 - **PCB (something) 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?

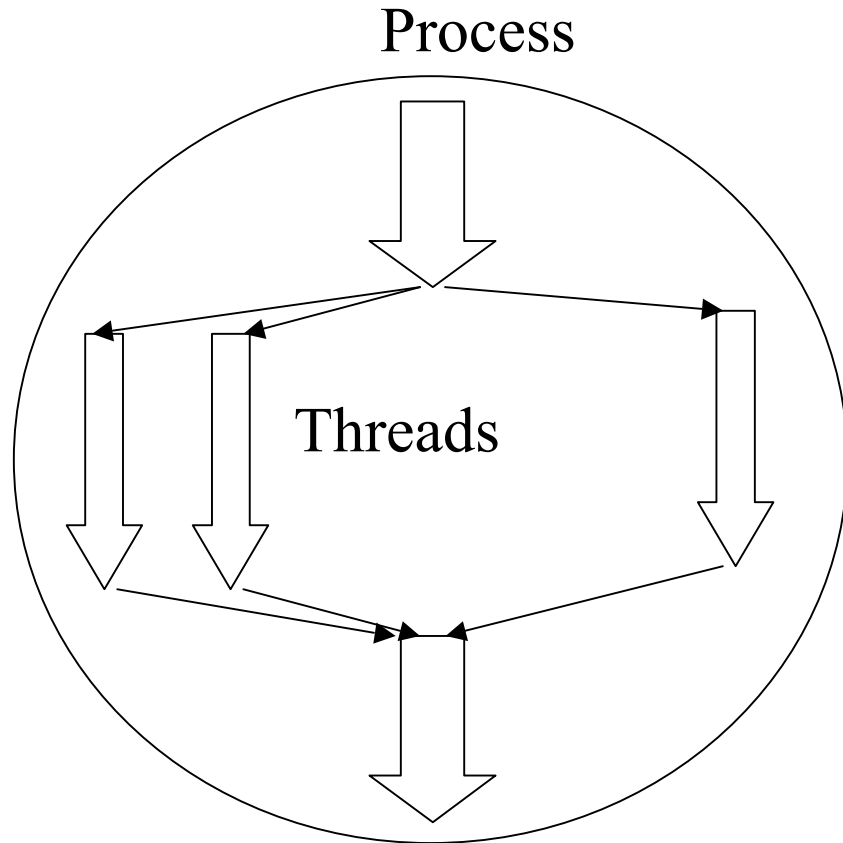
Win NT Idle

- No runnable 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;
- 

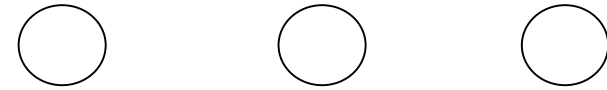
Threads and Processes

Trad. Threads

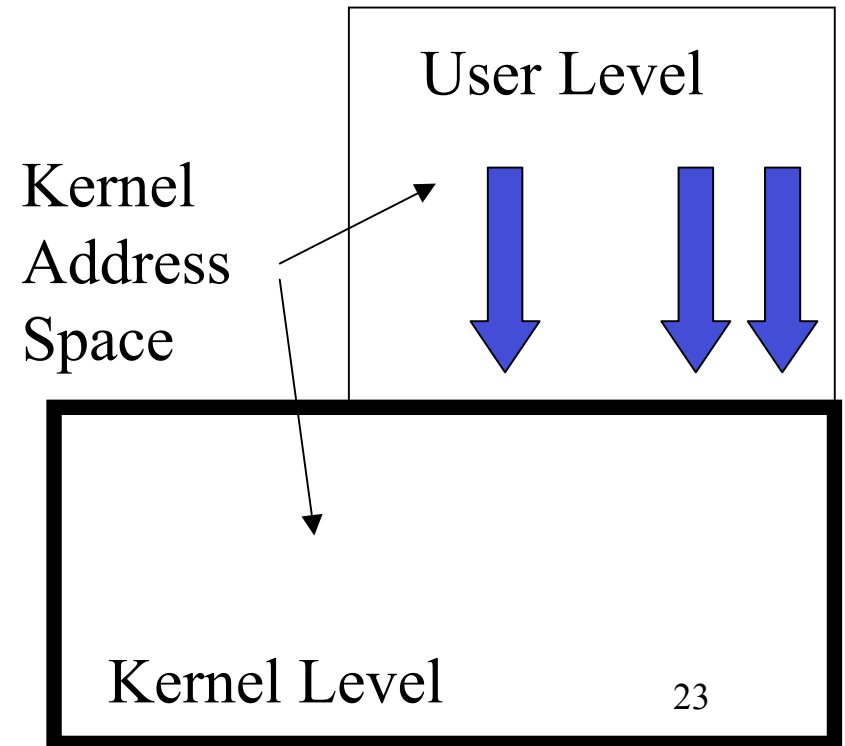
Project OpSys



Processes in individual address spaces



Kernel threads



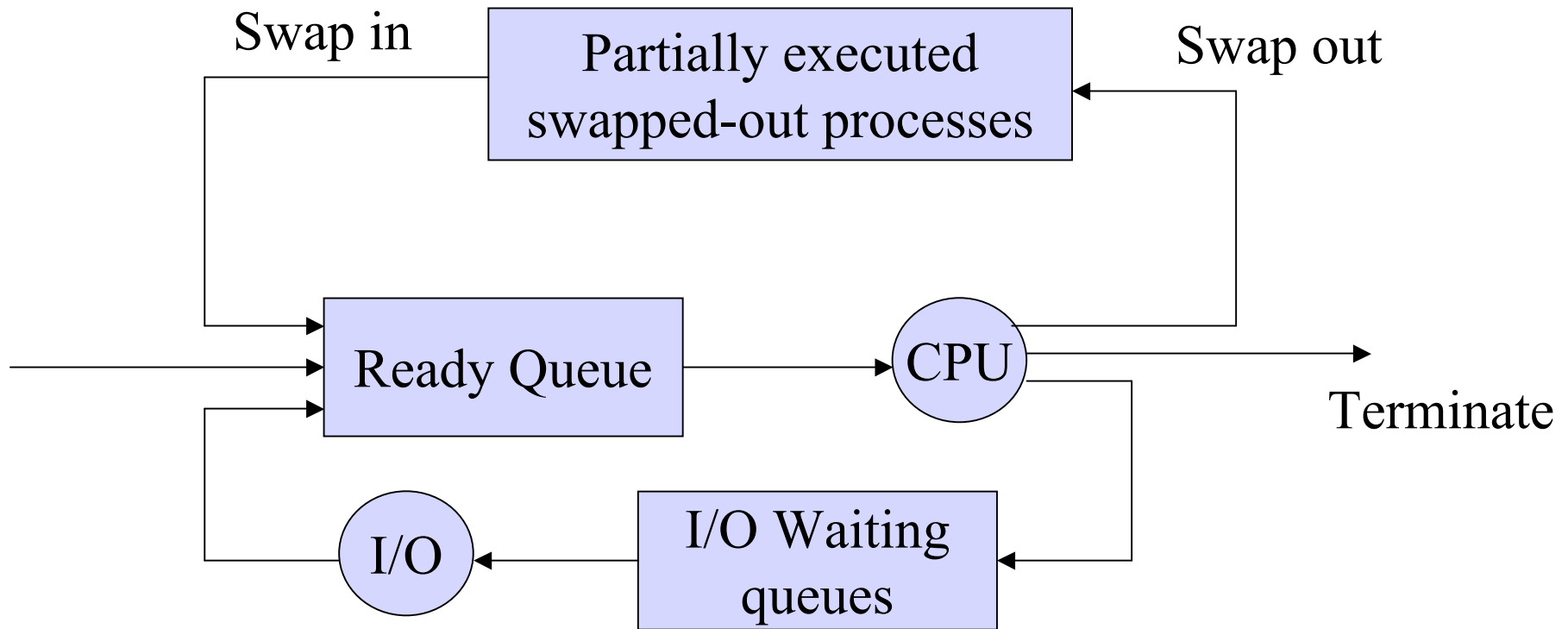
Where Should PCB Be Saved?

- 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

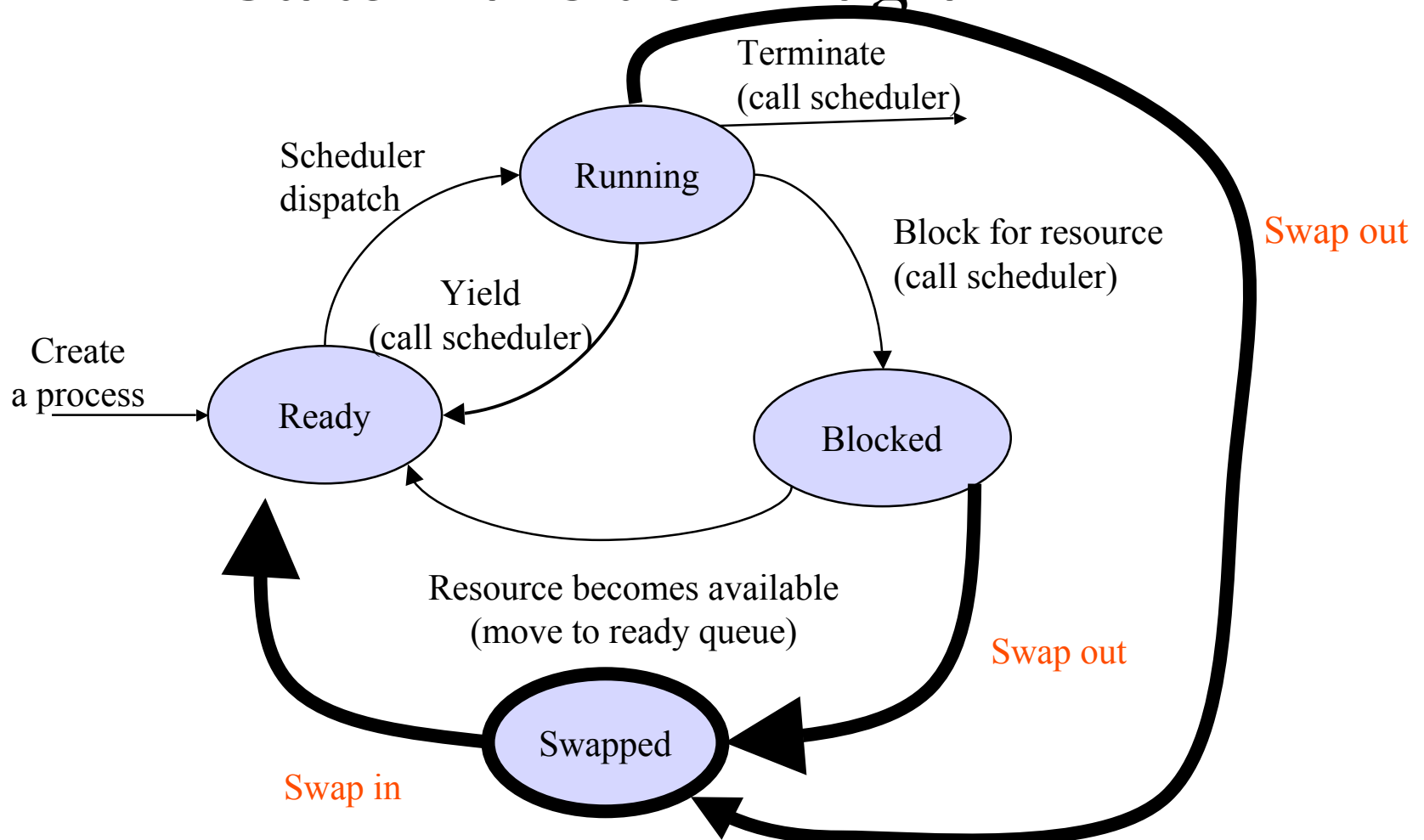
Job swapping

- 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
 - From which states(s), to which state(s)? *Try extending the following examples using two suspended states.*
- The term is also used in a slightly different setting, see MOS Ch. 4.2 pp. 196-197

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

I/O Multiplexing: More than one State Machine per Process

- `select` blocks for any of multiple events
- Handle (one of the events) that unblocks `select`
 - Advance appropriate state machine
- Repeat

Concurrent prog. w/ I/O Multiplexing

- Establishes several control flows (state machines) in one process
- Uses `select`
- Offers application programmer more control than processor model (How?)
- Easy sharing of data among state machines
- More efficient (no process switch to switch between control flows in same process)
- Difficult programming