

Processes and Non-Preemptive Scheduling

Tore Larsen
University of Tromsø

Otto J. Anshus
University of Tromsø, University of Oslo

Kai Li
Princeton University

An aside on concurrency

- Timing and sequencing of events are key concurrency issues
- We will study classical OS concurrency issues, including implementation and use of classic OS mechanisms to support concurrency
- In later courses on concurrency and on parallel programming you will revisit this material
- In a later course on distributed systems you may want to use formal tools to better understand and model timing and sequencing
- Emerging CPUs are typically multi core, and multi threaded
 - *Challenges and opportunities to design cost-effective high-performance systems.*

Process (I)

- 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)

- Classically, processes were, using today's terminology; “Single Threaded”
- Sequential program: Single process
- Parallel program: Multiple cooperating processes
 - This works fine in principle, but interaction among processes incurs significant OS time overhead.

Processes (III)

- “Modern” process: “Process” and “Thread” are separate 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

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

Supporting and Using Processes

- Multiprogramming
 - Supporting concurrent execution (*parallel or transparently interleaved*) of multiple processes (or threads), typically assumed to be related to different problems
 - Achieved by process- or context switching, switching the CPU(s) back and forth among the processes, keeping track of each process' progress
- Concurrent programs
 - Programs (or threads) that exploit multiprogramming for some purpose (e.g. performance, structure)
 - Operating systems is important application area for concurrent programming. Many others (event driven programs, servers, ++)
 - <http://csapp.cs.cmu.edu/public/ch12-preview.pdf>

Implementing processes

- OS needs to keep track of all processes
 - Keep track of it's progress
 - Parent process
 - 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 issued before corresponding exit: Parent blocks until exit
 - Exit issued before corresponding wait: Child becomes zombie (un-dead) until wait
- Kill, specified process terminates

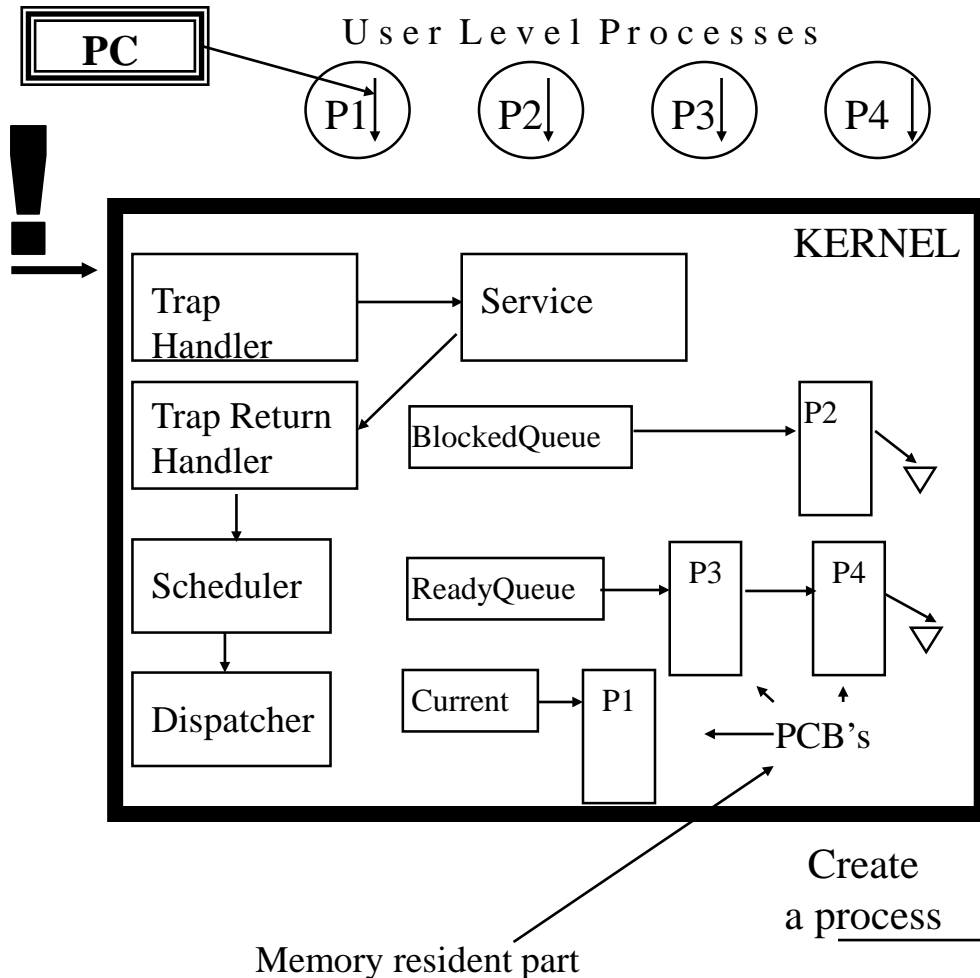
When may OS switch contexts?

- Only when OS runs
 - Events potentially causing a context switch:
 - Process created (`fork`)
 - Process exits (`exit`)
 - Process blocks implicitly (I/O, IPC)
 - Process blocks explicitly (`yield`)
 - User or System Level Trap
 - HW
 - SW: User level System Call
 - 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

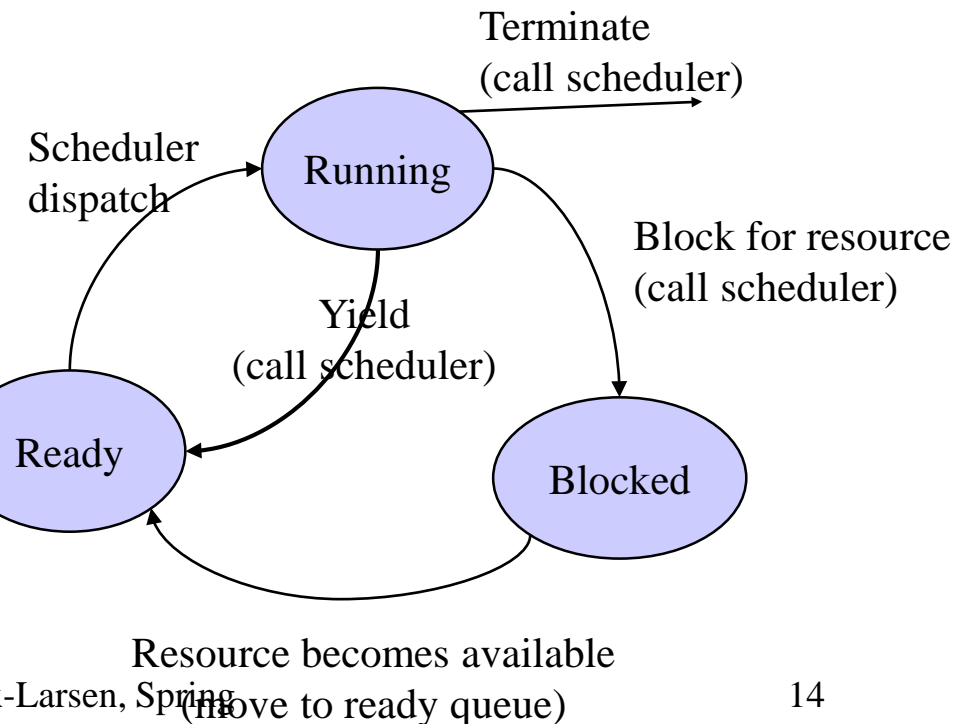
- Performance
 - Should be no more than a few microseconds
 - Most time is spent SAVING and RESTORING the context of processes
 - Less processor state to save, the better
 - Pentium has a multitasking mechanism, but SW can be faster if it saves less of the state
 - How to save time on the copying of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

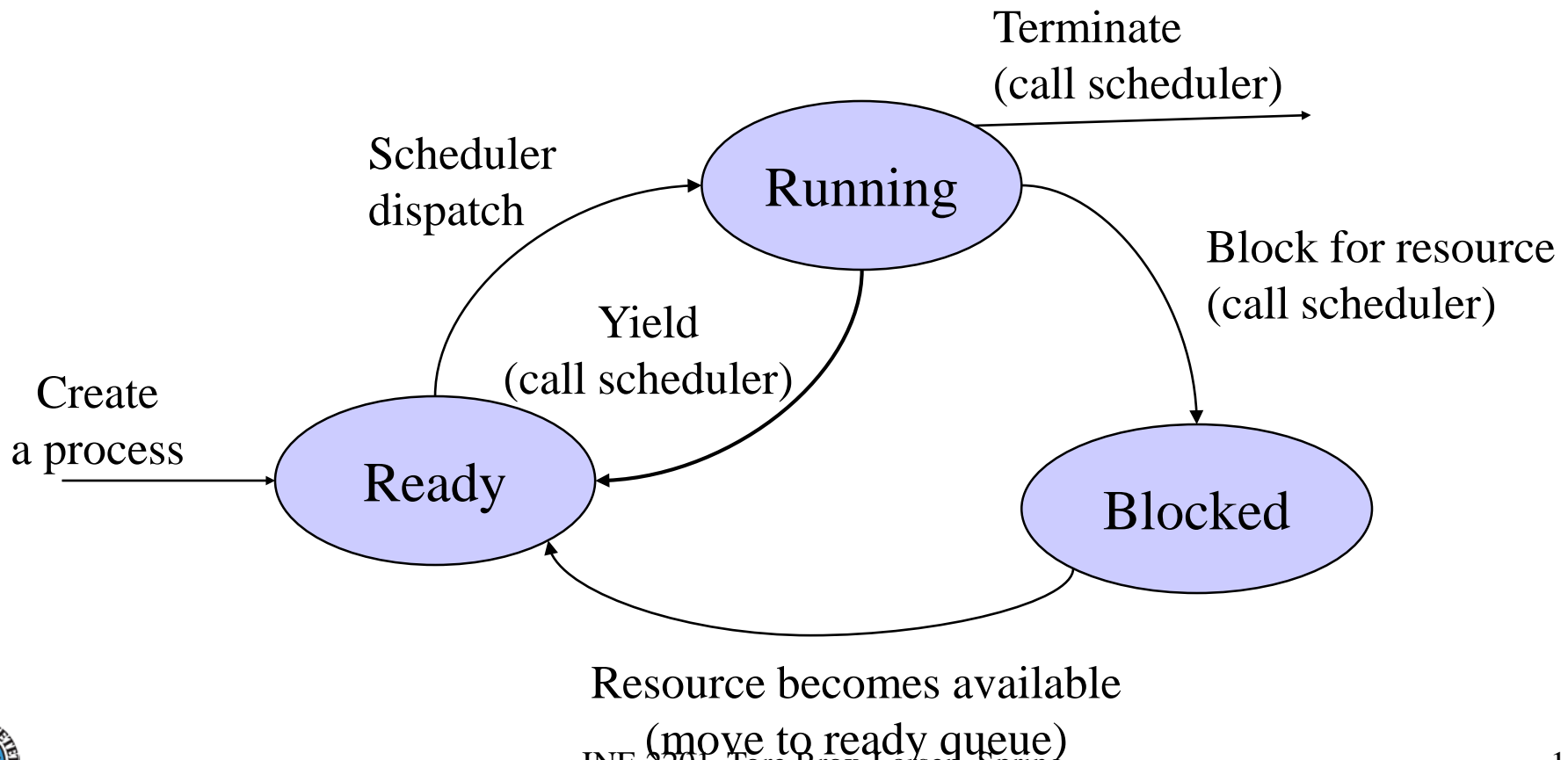


MULTIPROGRAMMING

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



Process State Transition of Non-Preemptive Scheduling



Scheduler

- Non-preemptive scheduler invoked by explicit block or yield calls, possibly also 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

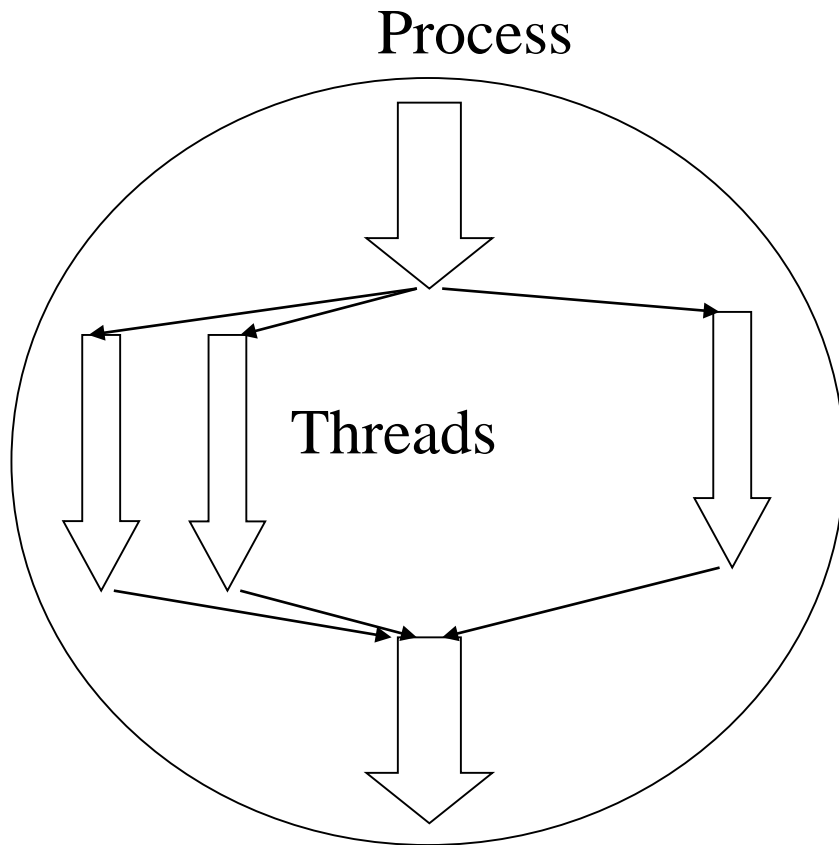
- Should the scheduler use a special stack?
 - Yes,
 - because a user process can overflow and it would require another stack to deal with stack overflow
 - because it makes it simpler to pop and push to rebuild a process's context
 - Must have a stack when booting...
- Should the scheduler simply be a “kernel process”?
 - You can view it that way because it has a stack, code and its data structure
 - This process 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;

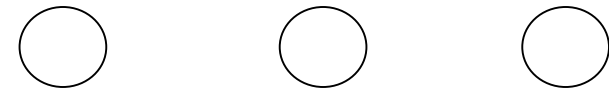
Threads and Processes

Trad. Threads

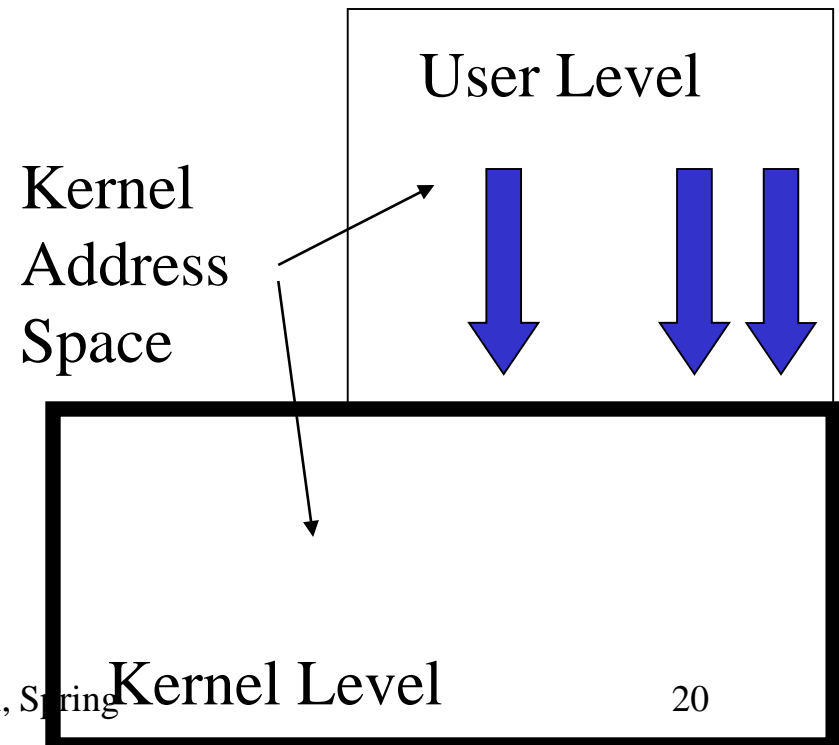


Project OpSys

Processes in individual address spaces



Kernel threads



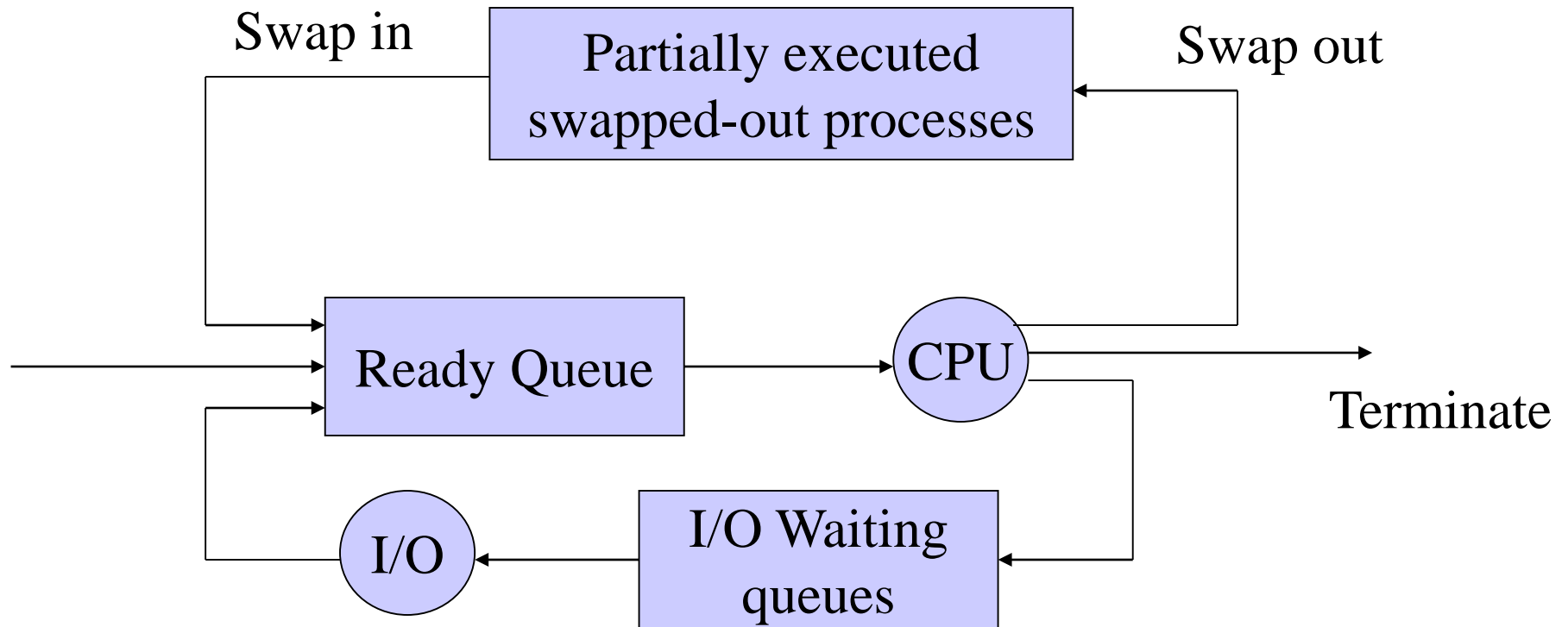
Where Should PCB Be Saved?

- 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

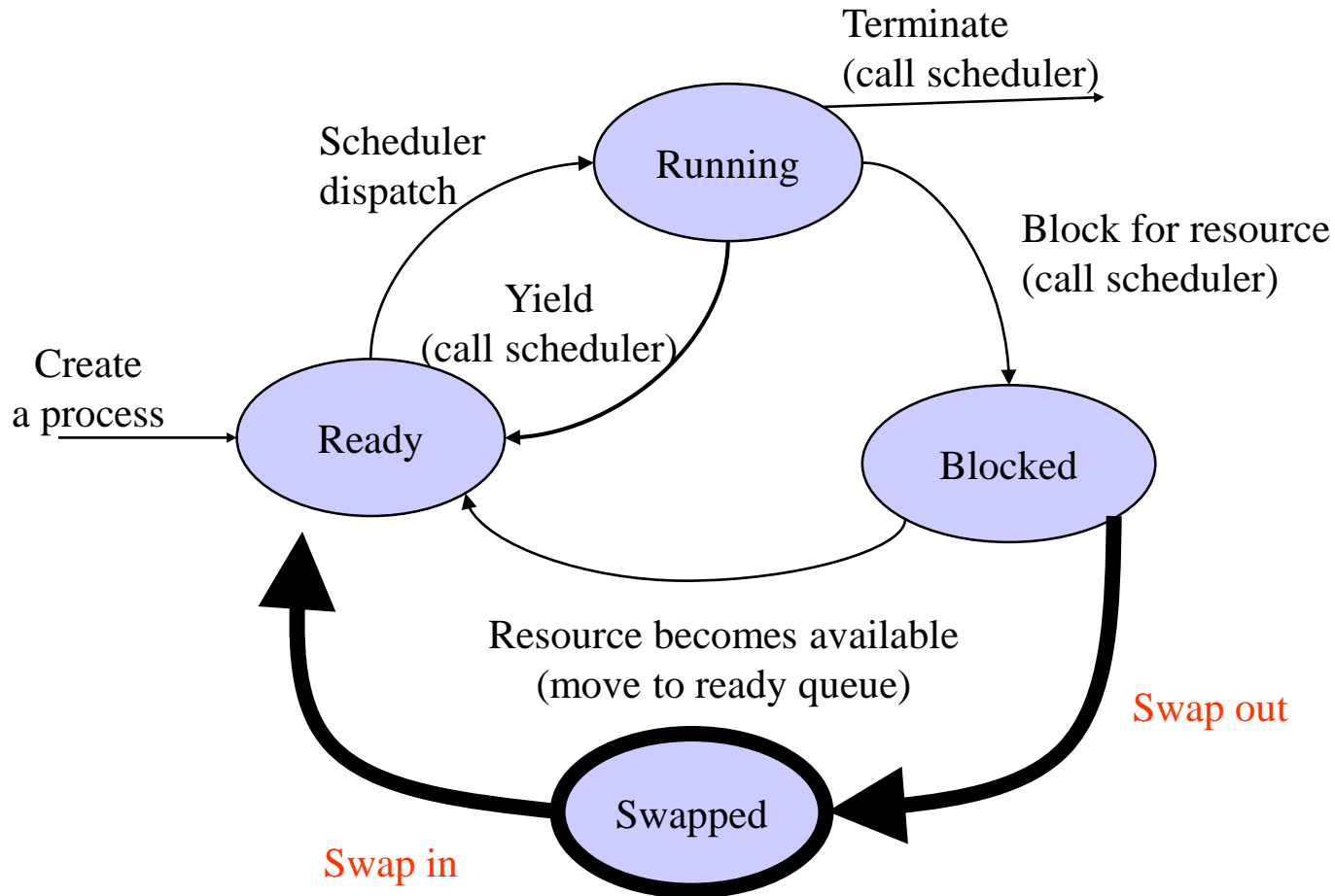
Job swapping

- 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.*
- Check&relate: MOS 3e.: Ch. 2.1.7, 3.2.1 and 3.3

Job Swapping, ii



Add Job Swapping to State Transition Diagram



Concurrent Programming w/ Processes

- Clean programming model
 - File tables are shared
 - User address space is private
 - Processes are protected from each other
 - Sharing requires some sort of IPC (InterProcess Communication)
- 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 specified I/O events
- Handle (one of the events) that unblocks `select`
 - Advance appropriate state machine
- Repeat
- See <http://csapp.cs.cmu.edu/public/ch12-preview.pdf>

Concurrent prog. w/ I/O Multiplexing

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