Processes, Threads, and Scheduling

Baskin Engineering
UC SANTA CRUZ

Processes and threads

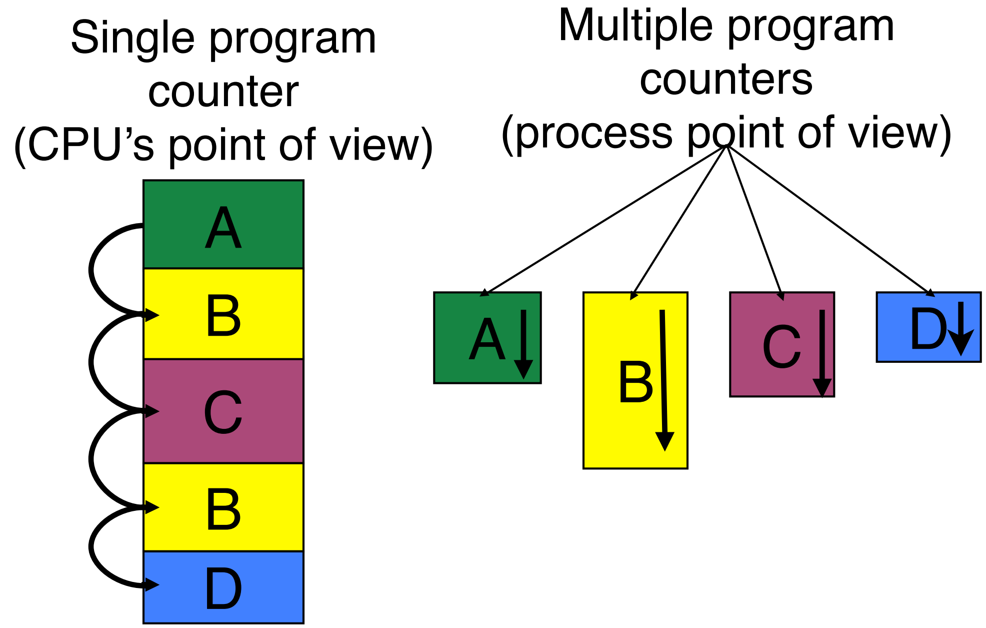
- Processes
- Threads
- Scheduling
- Interprocess communication (IPC)
- Classical IPC problems

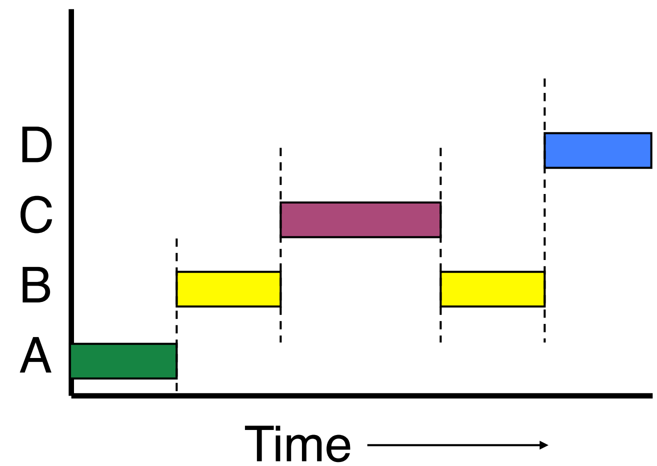
What is a process?

- Code, data, and stack
  - Usually (but not always) has its own address space
- Program state
  - CPU registers
  - Program counter (current location in the code)
  - Stack pointer
- Only one process can be running in the CPU at any given time!

The process model

- Multiprogramming of four programs
- Conceptual model
  - 4 independent processes
  - Processes run sequentially
- Only one program active at any instant!
  - That instant can be very short…
  - Only applies if there's a single CPU (with a single core) in the system





Baskin Engineering UC SANTA CRUZ

When is a process created?

- Processes can be created in two ways
  - System initialization: one or more processes created when the OS starts up
  - Execution of a process creation system call: something explicitly asks for a new process
- System calls can come from
  - User request to create a new process (system call executed from user shell)
  - Already running processes
    - User programs
    - System daemons

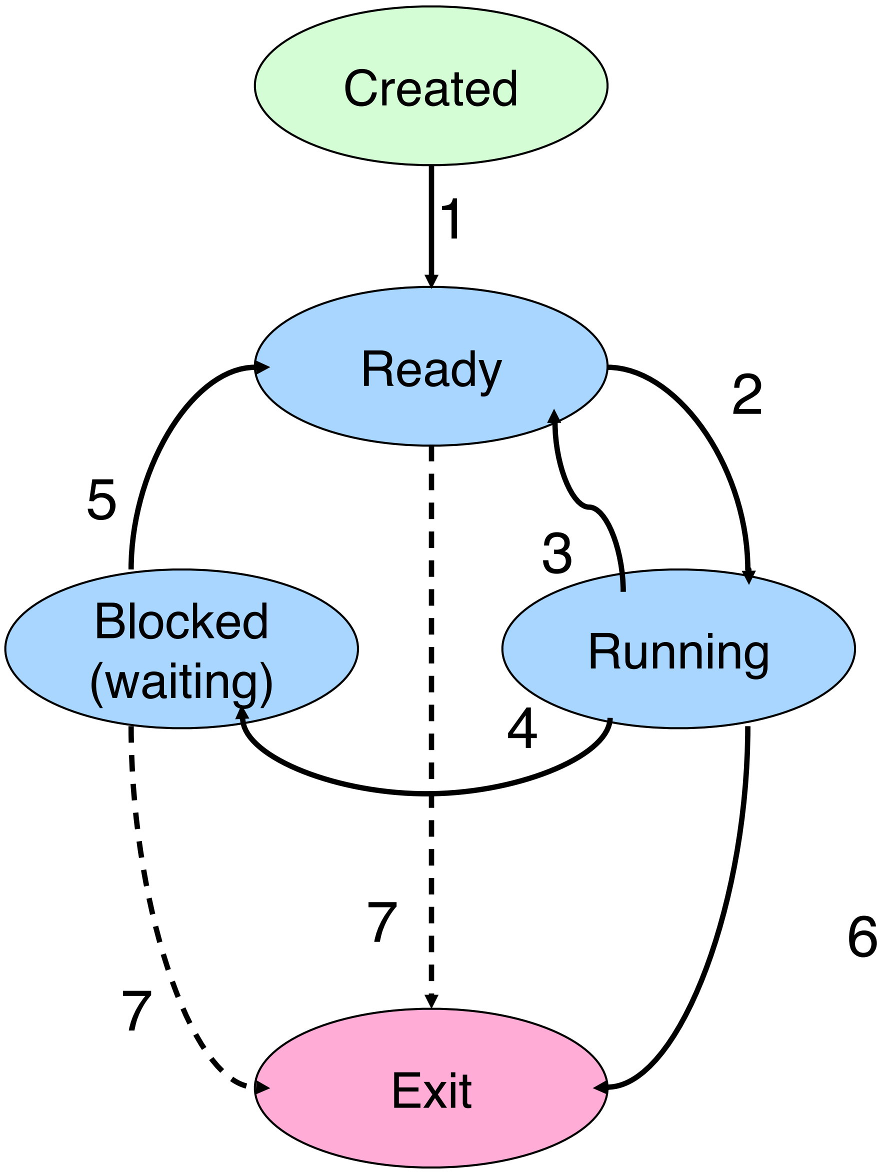
When do processes end?

- Conditions that terminate processes can be
  - Voluntary
  - Involuntary
- Voluntary
  - Normal exit
  - Error exit
- Involuntary
  - Fatal error (only sort of involuntary)
  - Killed by another process

Process hierarchies

- Parent creates a child process
  - Child processes can create their own children
- Forms a hierarchy
  - UNIX calls this a “process group”
  - If a process terminates, its children are “inherited” by the terminating process’s parent
- Windows has process groups
  - Multiple processes grouped together
  - One process is the “group leader”

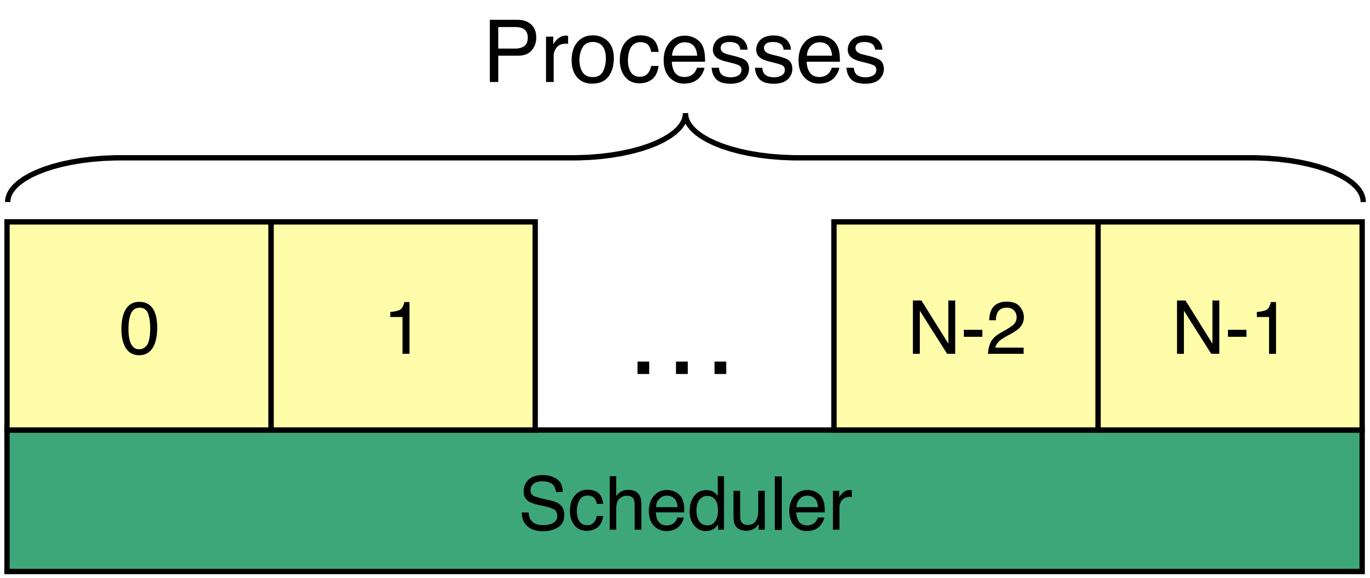
Process states



- Process in one of 5 states
  - Created
  - Ready
  - Running
  - Blocked
  - Exit
- Transitions between states
  - Process enters ready queue
  - Scheduler picks this process
  - Scheduler picks a different process
  - Process waits for event (such as I/O)
  - Event occurs
  - Process exits
  - Process ended by another process

Processes in the OS

- Two “layers” for processes
- Lowest layer of process-structured OS handles interrupts, scheduling
- Above that layer are sequential processes
  - Processes tracked in the process table
  - Each process has a process table entry



What's in a process table entry?

May be stored on stack

Process management

- Registers
- Program counter
- CPU status word
- Stack pointer
- Process state
- Priority / scheduling parameters
- Process ID
- Parent process ID
- Signals
- Process start time
- Total CPU usage

File management

- Root directory
- Working (current) directory
- File descriptors
- User ID
- Group ID

Memory management

Pointers to text, data, stack

or

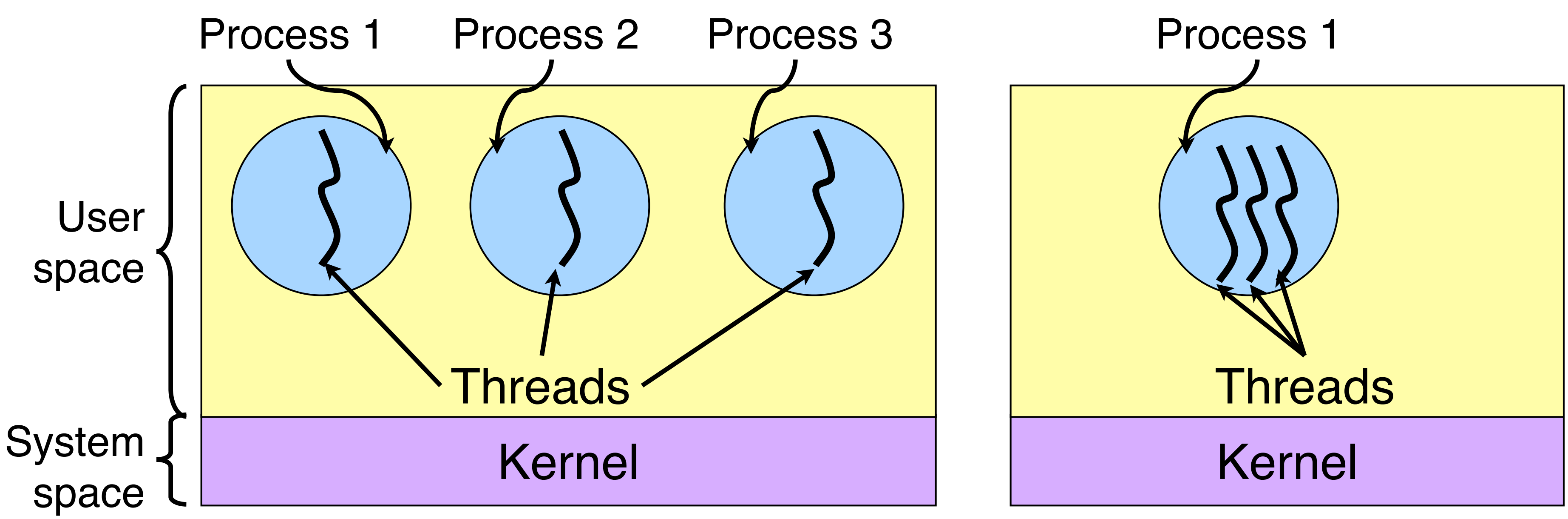
Pointer to page table

What happens on a trap/ interrupt?

1. Hardware saves program counter (on stack or in a special register)
2. Hardware loads new PC, identifies interrupt
3. Assembly language routine saves registers
4. Assembly language routine sets up stack
5. Assembly language calls C to run service routine
6. Service routine calls scheduler
7. Scheduler selects a process to run next (might be the one interrupted…)
8. Assembly language routine loads PC & registers for the selected process

Threads: “processes” sharing memory

- Process == address space
- Thread == program counter / stream of instructions
- Two examples
  - Three processes, each with one thread
  - One process with three threads



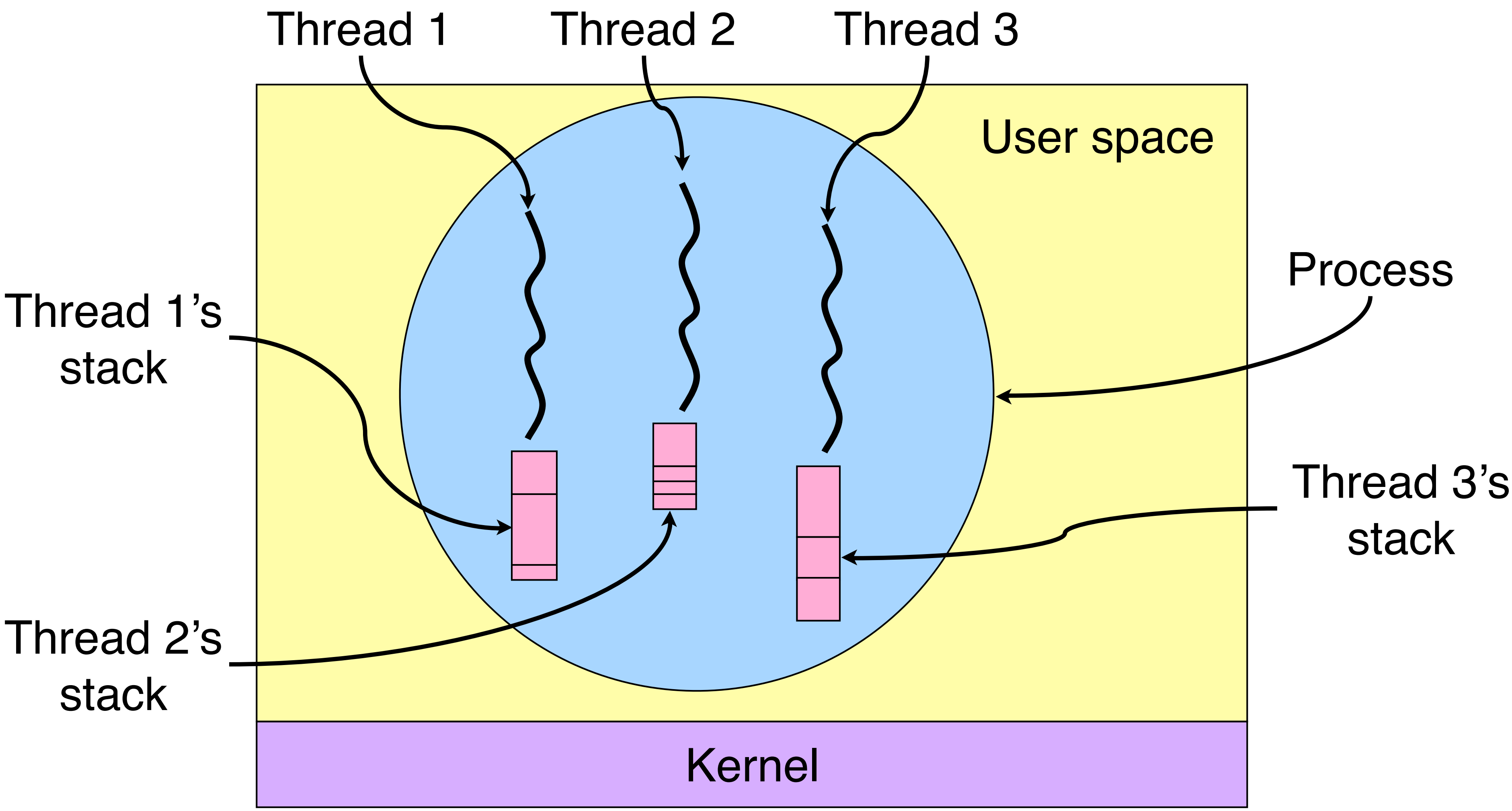
Process & thread information

Per process items

Address space
Open files
Child processes
Signals & handlers
Accounting info
Global variables

Per thread items	Per thread items	Per thread items
Program counter	Program counter	Program counter
Registers	Registers	Registers
Stack & stack pointer	Stack & stack pointer	Stack & stack pointer
State	State	State

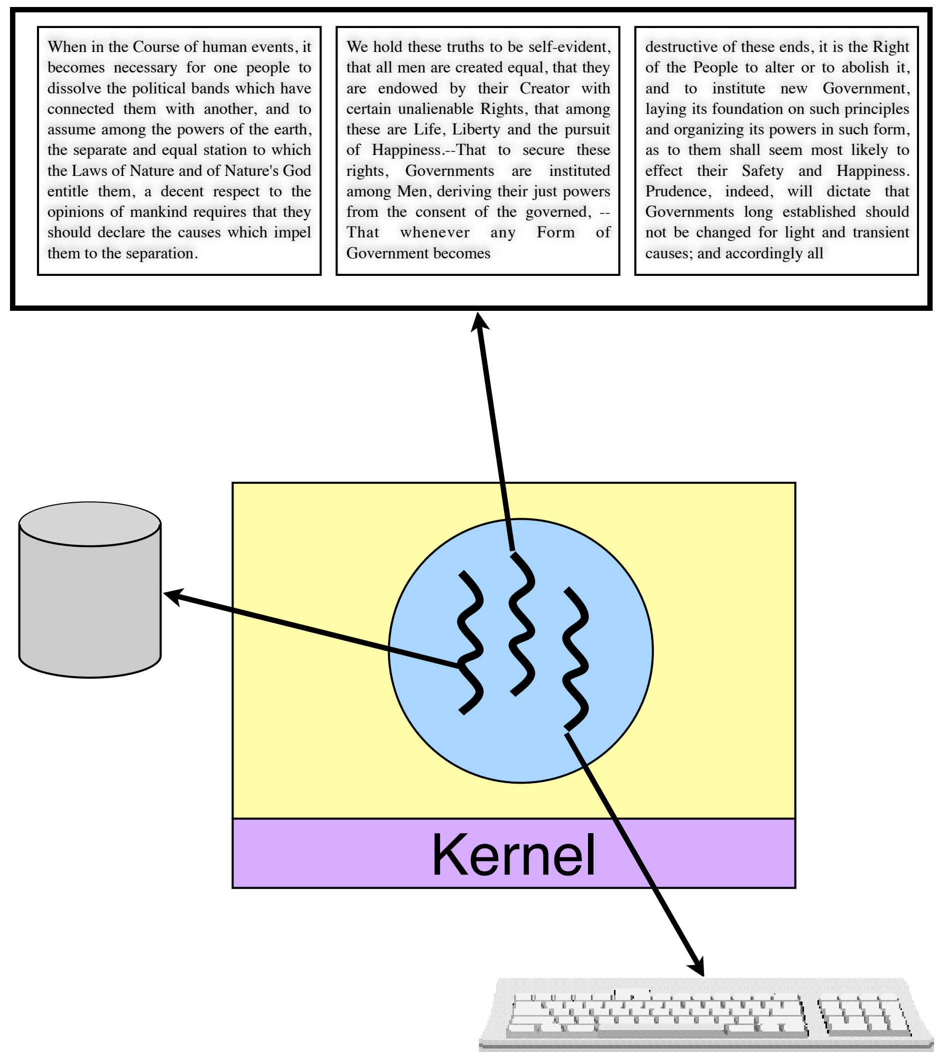
Threads & stacks



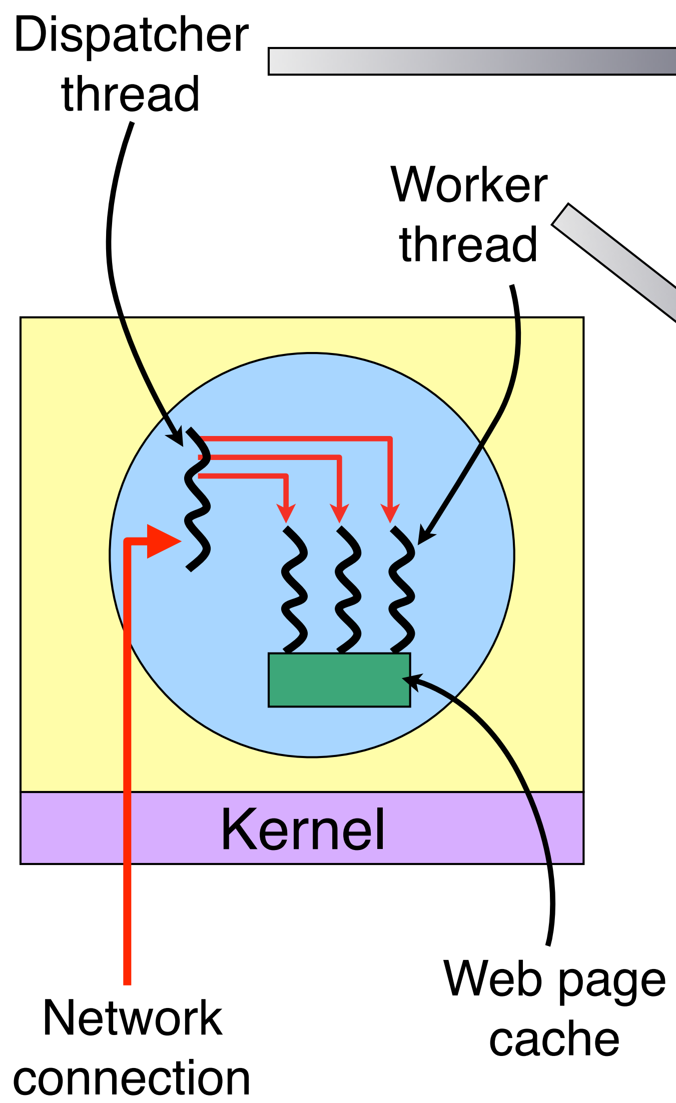
➡ Each thread has its own stack!

Why use threads?

- Allow a single application to do many things at once
  - Simpler programming model
  - Less waiting
- Threads are faster to create or destroy
  - No separate address space
- Overlap computation and I/O
  - Could be done without threads, but it's harder
- Example: word processor
  - Thread to read from keyboard
  - Thread to format document
  - Thread to write to disk



Multithreaded Web server




```
while(TRUE) {
  getNextRequest(&buf);
  handoffWork(&buf);
}
```

```
while(TRUE) {
  waitForWork(&buf);
  lookForPageInCache(&buf,&page);
  if(pageNotInCache(&page)) {
    readPageFromDisk(&buf,&page);
  }
  returnPage(&page);
}
```

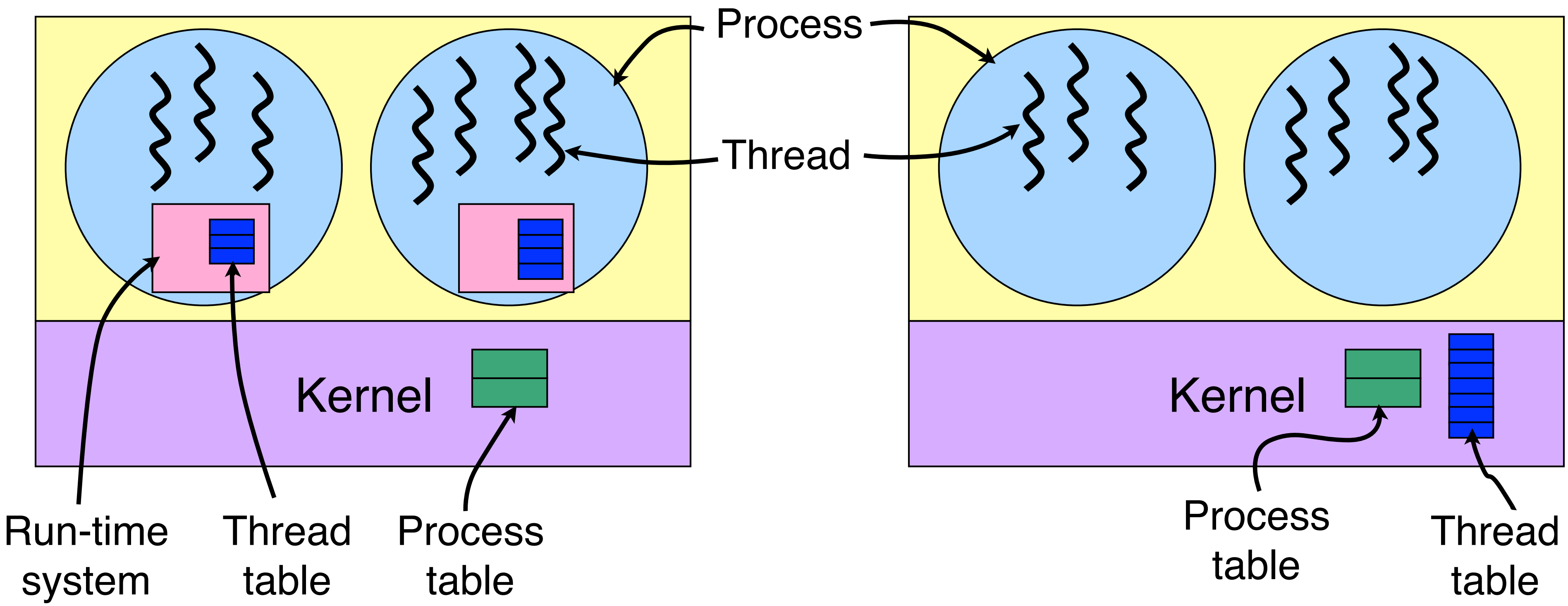
Three ways to build a server

- Multithreaded server
  - Parallelism
  - Blocking system calls
  - May use pop-up threads: create a new thread in response to an incoming message (rather than reusing a thread)
- Single-threaded process: slow, but easier to do
  - No parallelism
  - Blocking system calls
- Finite-state machine (event model)
  - Each activity has its own state: states change when system calls complete or interrupts occur
  - Parallelism
  - Nonblocking system calls

Issues with using threads

- May be tricky to convert single-threaded code to multithreaded code
- Re-entrant code
  - Code must function properly when multiple threads are using it simultaneously
  - Need to be careful when using static or global variables
    - Returned structures
    - Buffers
- Error management
  - What happens when just a single thread has an error?
  - Can’t simply kill the process, since other threads might be running

Implementing threads



User-level threads

+ No need for kernel support
- May be slower than kernel threads
- Harder to do non-blocking I/O

Kernel-level threads

+ More flexible scheduling
+ Non-blocking I/O
- Not portable

POSIX threads

- Standard interface to threading library
- May be implemented in either user or kernel space
  - Some operating systems provide support for both!
- Allows thread-based programs to be portable

Thread call (`Pthread_xx`)	Description
`create`	Create a new thread
`exit`	Terminate the calling thread
`join`	Wait for a specific thread to exit
`yield`	Release the CPU, allowing another thread to run

Processes & threads in Linux

- Linux supports kernel-level threads (lightweight processes)
  - Share address space, file descriptors, etc.
  - Each has its own process descriptor in memory
- Linux processes (incl. lightweight) all have unique identifiers
  - Threads sharing address space are grouped into *process groups*
  - Identifier shared by the group is that of the *leader*

Linux process information

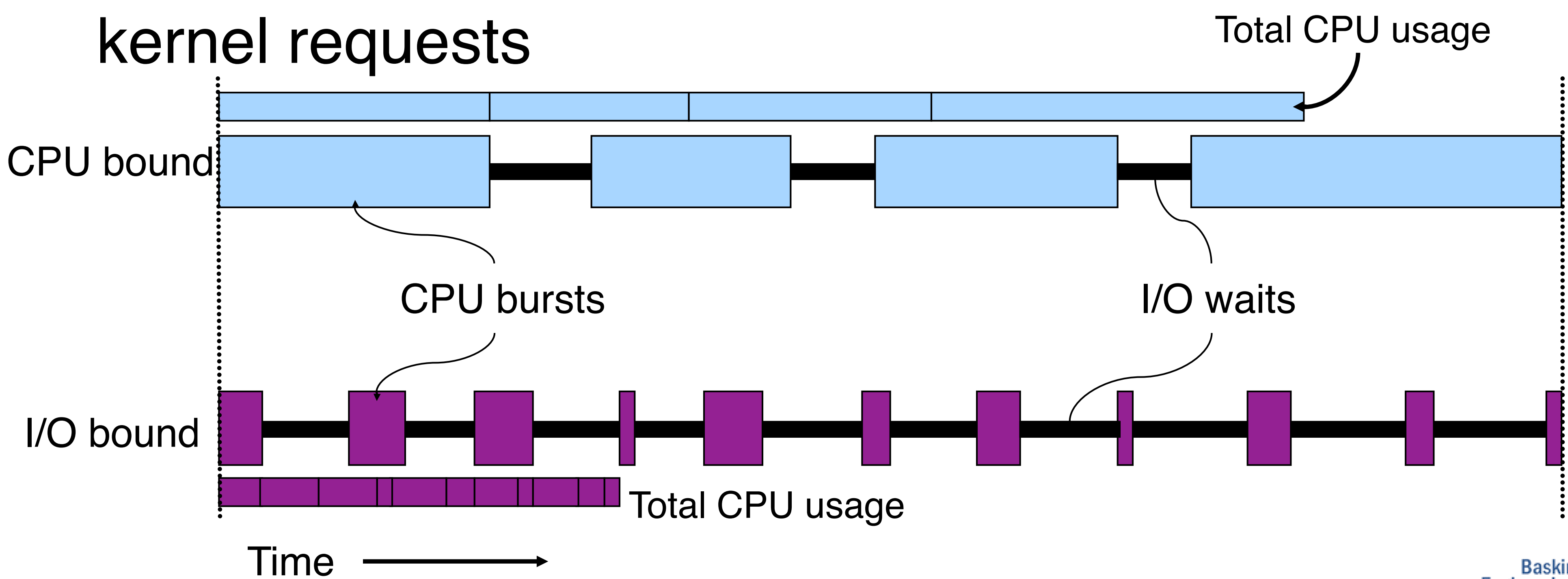
- Each process has its own 8KB region that stores
  - Kernel stack
    - Kernel has a small stack: about 4KB!
  - Low-level thread information
- Other information stored in a separate data structure
  - Memory allocated to the process
  - Open files
  - Signal information

Scheduling

- What is scheduling?
  - Goals
  - Mechanisms
- Scheduling on batch systems
- Scheduling on interactive systems
- Other kinds of scheduling
  - Real-time scheduling

Why schedule processes?

- Bursts of CPU usage alternate with periods of I/O wait
- Some processes are CPU-bound: they don't many I/O requests
- Other processes are I/O-bound and make many kernel requests



When are processes scheduled?

- At the time they enter the system
  - Common in batch systems
  - Two types of batch scheduling
    - Submission of a new job causes the scheduler to run
    - Scheduling only done when a job voluntarily gives up the CPU (i.e., while waiting for an I/O request)
- At relatively fixed intervals (clock interrupts)
  - Necessary for interactive systems
  - May also be used for batch systems
  - Scheduling algorithms at each interrupt, and picks the next process from the pool of “ready” processes

Scheduling goals

- All systems
  - Fairness: give each process a fair share of the CPU
  - Enforcement: ensure that the stated policy is carried out
  - Balance: keep all parts of the system busy
- Batch systems
  - Throughput: maximize jobs per unit time (hour)
  - Turnaround time: minimize time users wait for jobs
  - CPU utilization: keep the CPU as busy as possible
- Interactive systems
  - Response time: respond quickly to users' requests
  - Proportionality: meet users' expectations
- Real-time systems
  - Meet deadlines: missing deadlines is a system failure!
  - Predictability: same type of behavior for each time slice

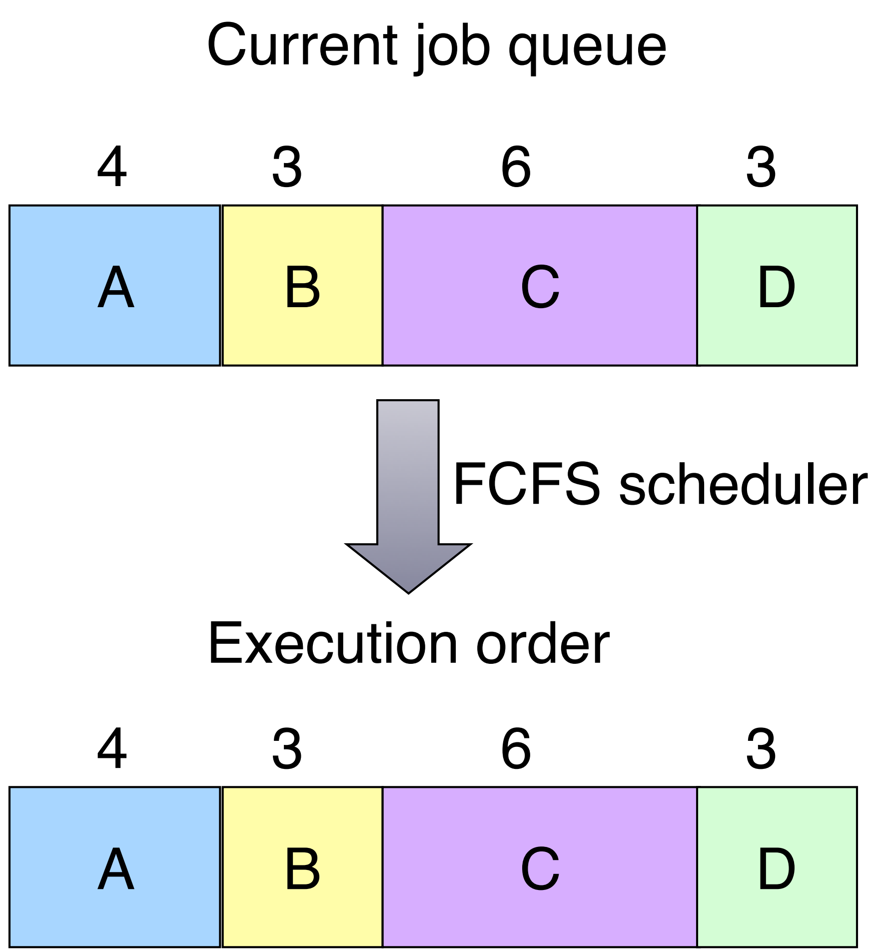
Measuring scheduling performance

- Throughput
  - Amount of work completed per second (minute, hour)
  - Higher throughput usually means better utilized system
- Response time
  - Response time is time from when a command is submitted until results are returned
  - Can measure average, variance, minimum, maximum, …
  - May be more useful to measure time spent waiting
  - Can also measure how often response time is faster than a given time (e.g., 100 ms): useful for real-time systems and servers
- Turnaround time
  - Like response time, but for batch jobs (response is the completion of the process)
- Usually not possible to optimize for ***all*** metrics with a single scheduling algorithm

Interactive vs. batch scheduling

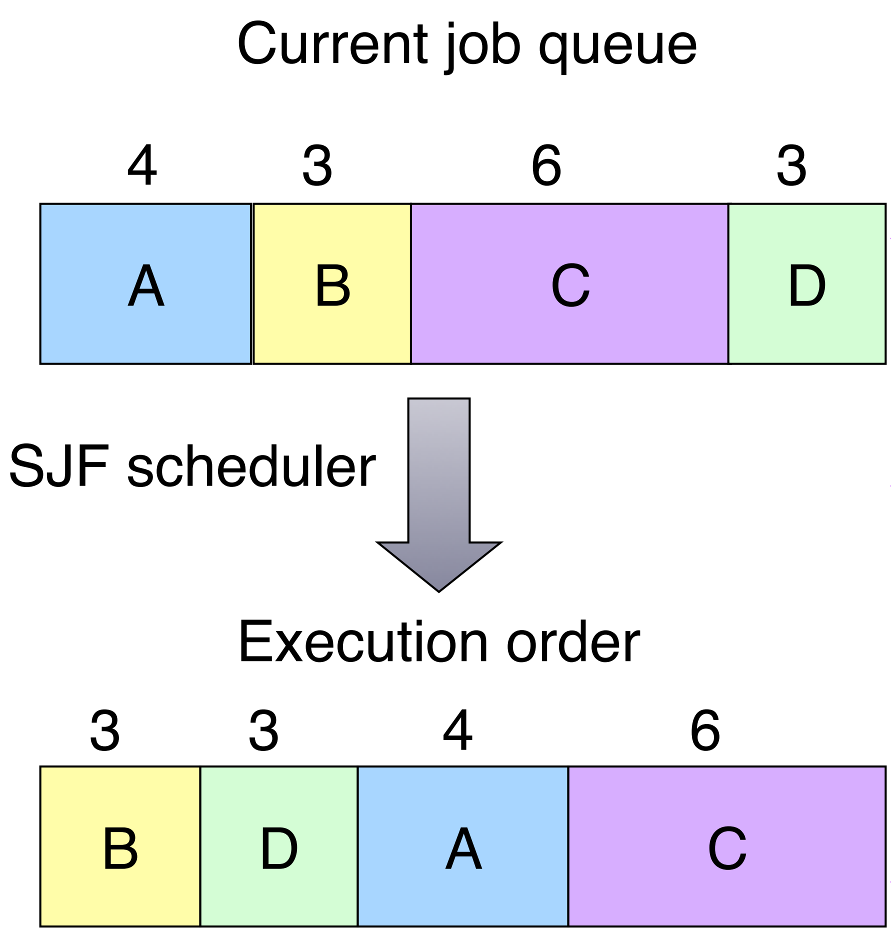
Batch	**Interactive**
First-Come-First-Served (FCFS)	Round-Robin (RR)
Shortest Job First (SJF)	Priority (preemptive)
Shortest Remaining Time First (SRTF)	Multi-level feedback queue
Priority (non-preemptive)	Lottery scheduling

First Come, First Served (FCFS)



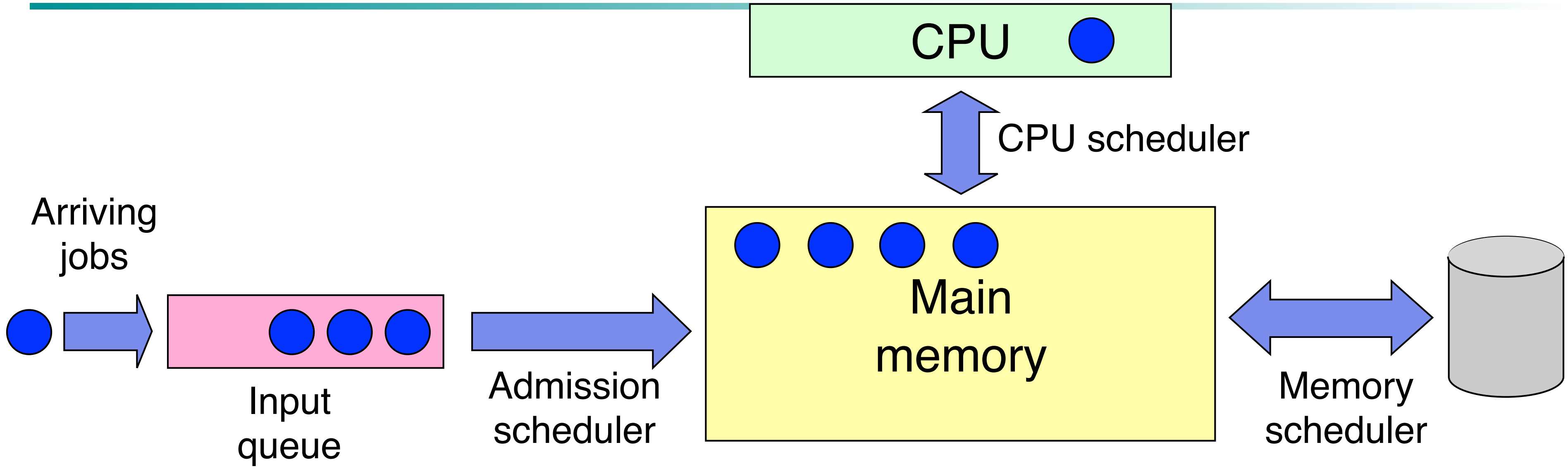
- Goal: do jobs in the order they arrive
  - Fair in the same way a bank teller line is fair
- Simple algorithm!
- Problem: long jobs delay every job after them
  - Many processes may wait for a single long job

Shortest Job First (SJF)



- Goal: do the shortest job first
  - Short jobs complete first
  - Long jobs delay every job after them
- Jobs sorted in increasing order of execution time
  - Ordering of ties doesn't matter
- Shortest Remaining Time First (SRTF): preemptive form of SJF
  - Re-evaluate when a new job is submitted
- Problem: how does the scheduler know how long a job will take?

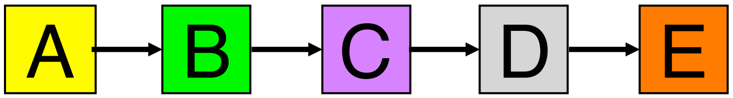
Three-level scheduling

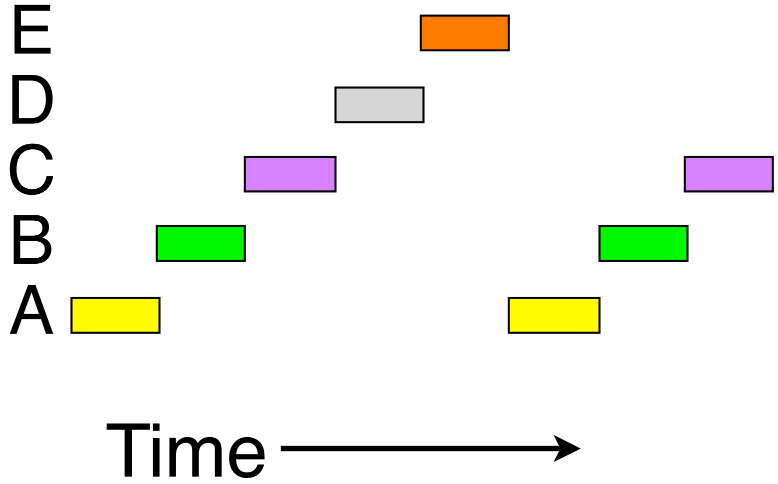


- Jobs held in input queue until moved into memory
  - Pick “complementary jobs”: small & large, CPU- & I/O-intensive
  - Jobs move into memory when admitted
- CPU scheduler picks next job to run
- Memory scheduler picks some jobs from main memory and moves them to disk if insufficient memory space

Round Robin (RR) scheduling

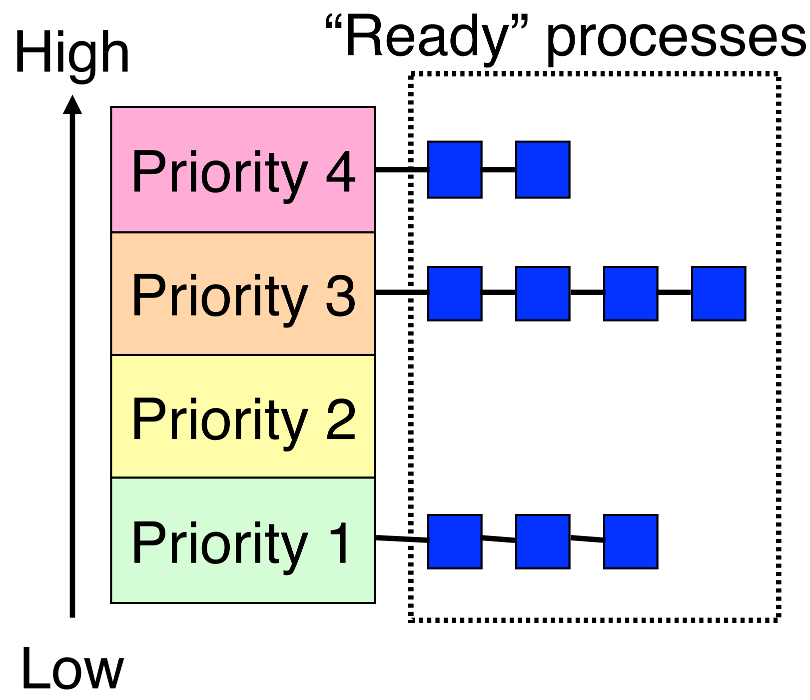
- Scheduling interactive processes
  - Give each process a fixed time slot (quantum)
  - Rotate through “ready” processes
  - Each process makes some progress
- What’s a good quantum?
  - Too short: many process switches hurt efficiency
  - Too long: poor response to interactive requests
  - Typical length: 10–100 ms
- “Strict” rotation: round robin





Priority scheduling

- Assign a priority to each process
  - "Ready" process with highest priority allowed to run
  - Running process may be interrupted after its quantum expires
- Priorities may be assigned dynamically
  - Reduced when a process uses CPU time
  - Increased when a process waits for I/O
- Often, processes grouped into multiple queues based on priority, and run round-robin per queue



Shortest process next

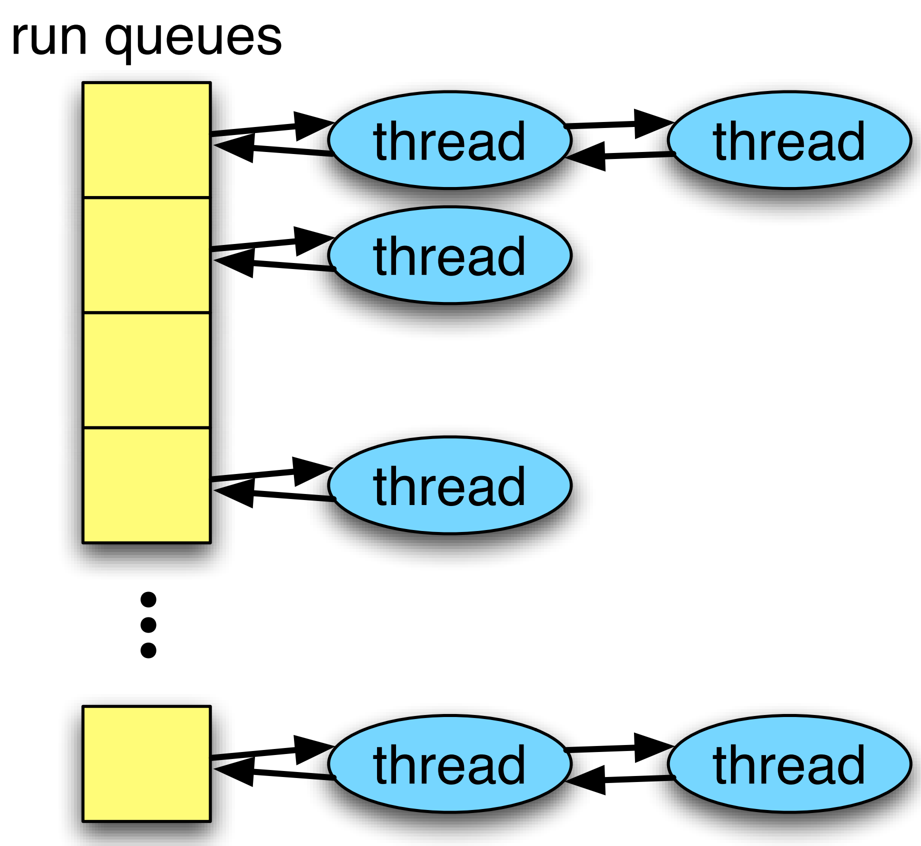
- Run the process that will finish the soonest
  - In interactive systems, job completion time is unknown!
- Guess at completion time based on previous runs
  - Update estimate each time the job is run
  - Estimate is a combination of previous estimate and most recent run time
- Not often used because round robin with priority works so well!

Lottery scheduling

- Give processes “tickets” for CPU time
  - More tickets ➡ higher share of CPU
- Each quantum, pick a ticket at random
  - If there are *n* tickets, pick a number from 1 to *n*
  - Process holding the ticket gets to run for a quantum
- Over the long run, each process gets the CPU *m/n* of the time if the process has m of the n existing tickets
- Tickets can be transferred
  - Cooperating processes can exchange tickets
  - Clients can transfer tickets to a server so it can have a higher priority

Scheduling in FreeBSD

- Quantum is 100 ms: longest that's OK for interactive scheduling
- Scheduler is based on multi-level feedback queues
  - Priority is based on two things
    - Resource requirements: blocked threads have higher priority when rescheduled
    - Previous CPU usage: CPU hogs have lower priority
- Each thread is placed into a run queue for its priority
  - Head of highest-priority run queue with a ready thread runs next



Calculating priority in FreeBSD

- Thread priority is set by:
  pri = MIN+[*estcpu*/4]+2×*nice*
  - Values above MAX are set to MAX
  - MIN=160, MAX=223
  - *nice* is set by the user to manually lower thread priority
  - *estcpu* is an estimate of the number of “ready” processes in the CPU when the calculation is made
    - Has a bit of “memory” so it doesn’t change too quickly
    - *estcpu* is updated each clock tick
  - Higher numbers indicate lower priority: threads with lowest priority values are scheduled first
- Thread priority is set every 40 ms
- Scheduling is more complex for multiprocessors...

Scheduling in Linux

- Three classes of processes
  - Conventional
  - Real-time (round-robin)
  - Real-time (FIFO)
- Queue structure similar to BSD
  - Two sets of queues (0–99 real-time, 100–139 conventional): *active* and *expired*
  - Scheduler runs process in lowest-valued *active* queue
  - Conventional threads placed in *expired* queue when their quantum is up
    - May be scheduled out before quantum expires: put back into *active* queue
  - Real-time threads placed back into *active* queue

Linux: rescheduling processes

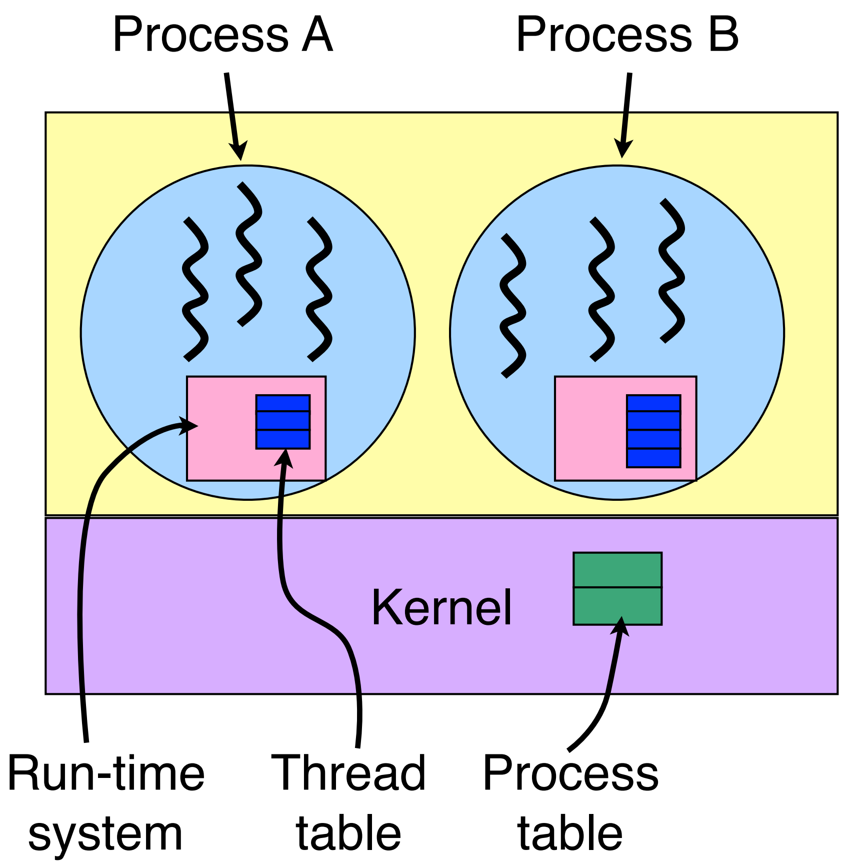
- Re-evaluate priorities when there are no threads on the *active* queue
- Thread priority is re-evaluated based on previous run time and sleep time
  - Threads that ran more and slept less are penalized with a worse priority
- Quantum is regenerated based on original (static) priority
  - Higher-priority threads are given longer quanta
  - Process may lose the CPU before quantum is up (if a higher-priority thread becomes available)

Further details in *Understanding Linux Kernel Internals (3rd edition)*, Bovet & Cesati

Policy versus mechanism

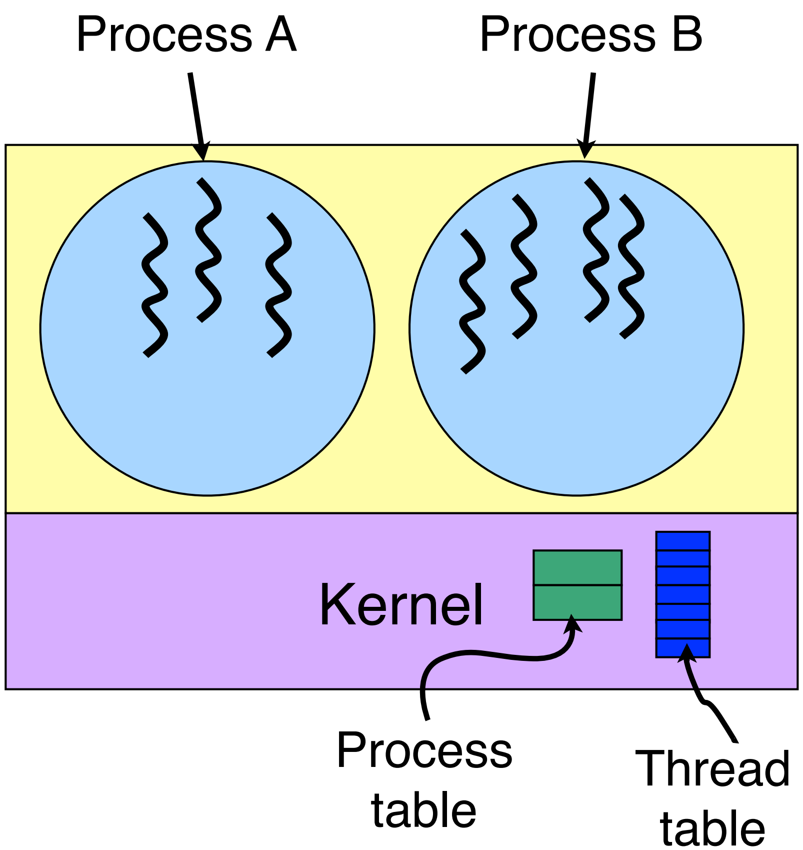
- Separate what may be done from how it is done
  - Mechanism allows
    - Priorities to be assigned to processes
    - CPU to select processes with high priorities
  - Policy set by what priorities are assigned to processes
- Scheduling algorithm parameterized
  - Mechanism in the kernel
  - Priorities assigned in the kernel or by users
- Parameters may be set by user processes
  - Don't allow a user process to take over the system!
  - Allow a user process to voluntarily lower its own priority
  - Allow a user process to assign priority to its threads

Scheduling user-level threads



- Kernel picks a process to run next
- Run-time system (at user level) schedules threads
  - Run each thread for less than process quantum
  - Example: processes get 40ms each, threads get 10ms each
- Example schedule: A1,A2,A3,A1,B1,B3,B2,B3
- Not possible: A1,A2,B1,B2,A3,B3,A2,B1

Scheduling kernel-level threads



- Kernel schedules each thread
  - No restrictions on ordering
  - May be more difficult for each process to specify priorities
- Example schedule: A1,A2,A3,A1, B1,B3,B2,B3
- Also possible: A1,A2,B1,B2, A3,B3,A2,B1