Processes and Threads

- What is a process? What is a thread? What types?
- A program has one or more locus of execution. Each execution is called a *thread* of execution. The set of threads comprise a *process*.
- Not an object or executable files: must be executing
- Each thread contains:
  - an **instruction pointer** (IP), a register with next instruction.
  - a **stack** for temporary data (eg, return addresses, parameters)
  - a **data area** for data declared globally and statically
- A process/thread is **active**, while a program is not.

How to run a program

- The executable code is loaded onto memory (where?)
- Space is allocated to variables (what types of vars?)
- A stack is allocated to the process (for what?)
- Registers are updated (which registers?)
- Control (of execution) goes to the process (how?)
- Process runs one instruction at a time, in a cycle:
  - fetch the instruction from memory
  - decode the instruction
  - update the IP
  - execute the instruction

Processes and Threads (revisited)

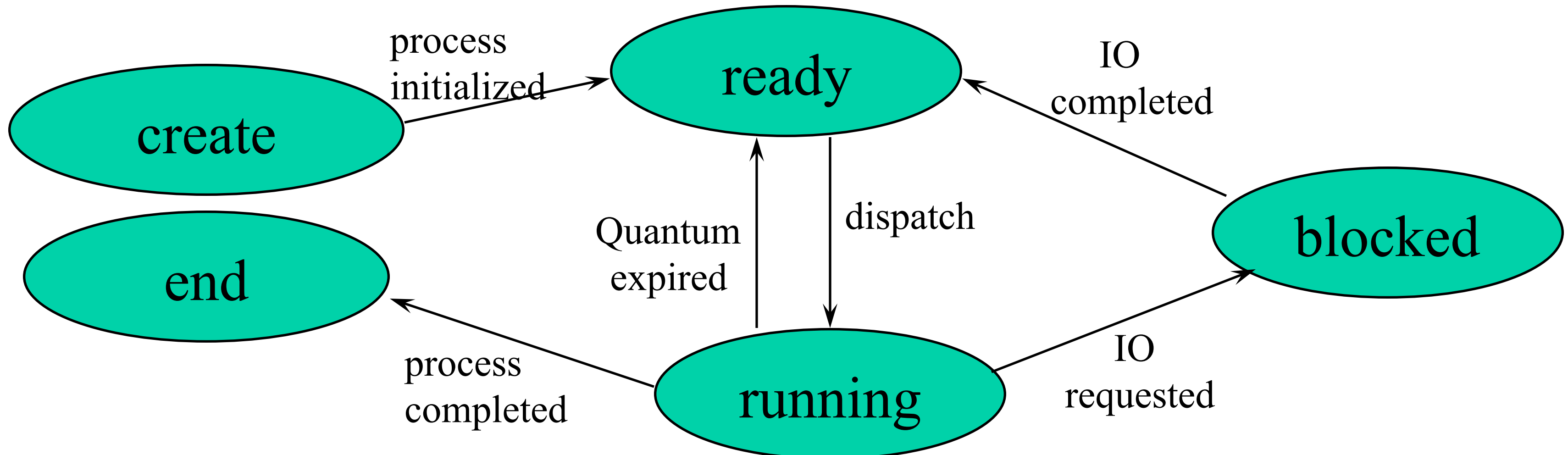
- *Address* space: memory reserved for a process
- A heavyweight process has a single locus of execution per address space, a single IP, a single PCB
- A *Process Control Block* (PCB) contains information pertaining to the process itself
  - state (running, ready, etc)
  - registers and flags (stack pointer, IP, etc)
  - resource information (memory/CPU usage, open files, etc)
  - process ID
  - security and protection information
  - accounting info (who to bill, limits, similar to resource info)

Processes and Threads (cont)

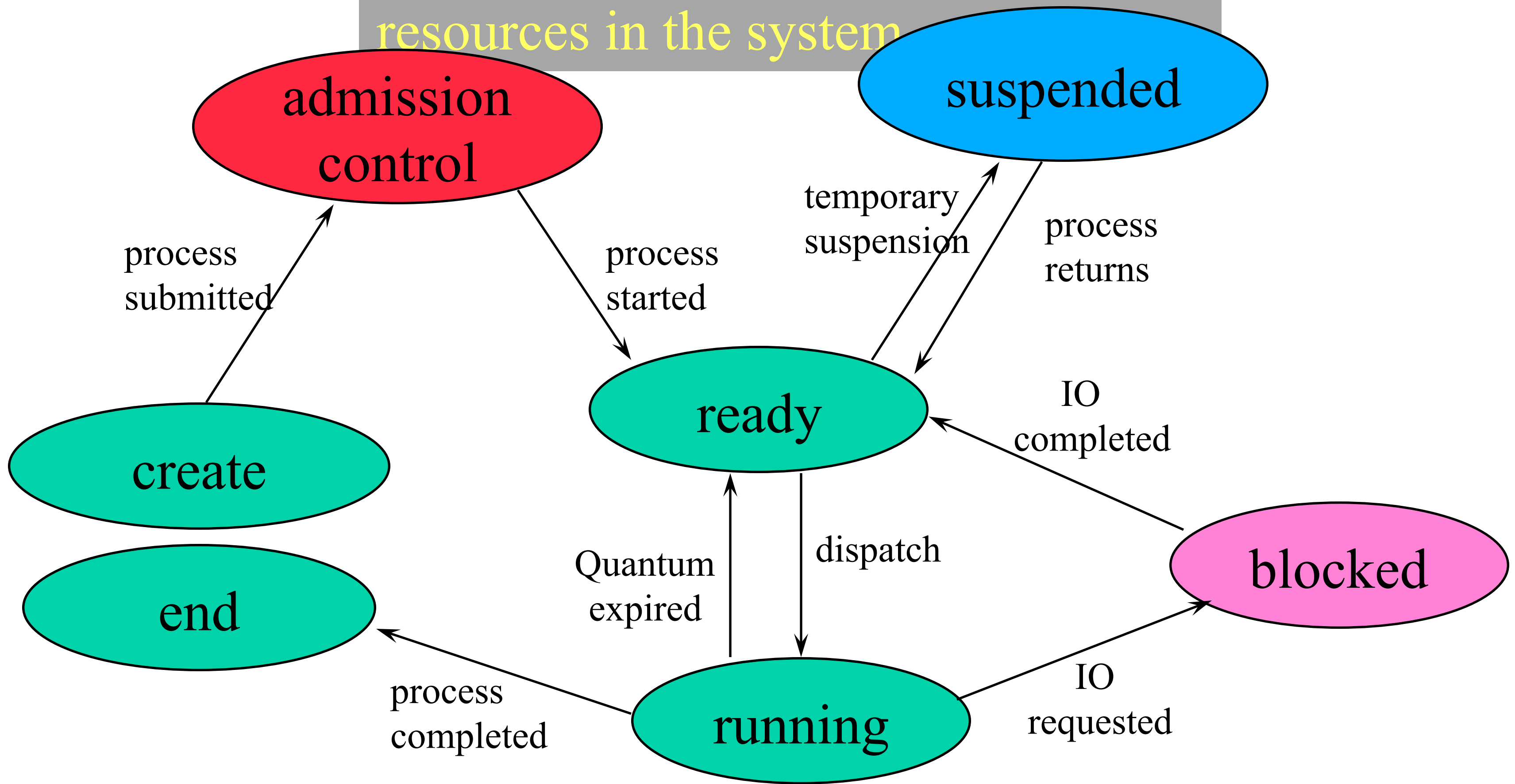
- Thread (*lightweight process*) of a process share some resources (e.g., memory, open files, etc)
- Threads have their own stack, IP, local address space
- With only a single process in memory, easy to manage
- For example, if a process requests IO (eg, read from keyboard), it just stays in the memory waiting
- Several threads or processes complicate things; when IO is requested, why make other processes wait?
- *Context switching* takes place… take a waiting thread "out of the CPU" and put a thread that is ready in CPU

State Diagram

- Create: PCB and other resources are setup
- End: resources held are returned to the OS (freed)
- Context switching: saves HW context; updates PCB
- States are typically implemented as queues, lists, sets



Complete State Diagram

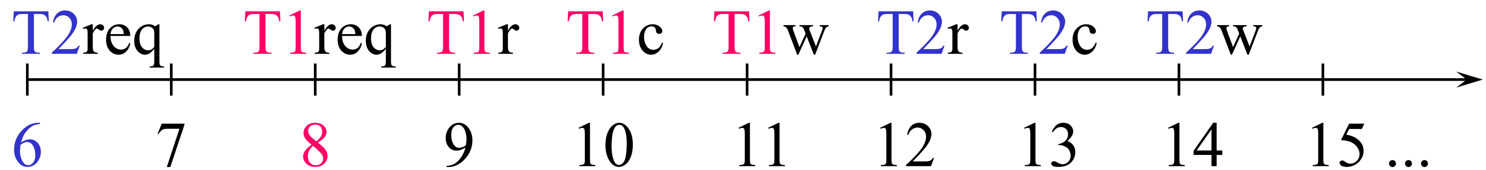


Multiple Threads and Processes

- Several problems with multitasking:
  - fairness in usage of CPU
  - fairness in usage of other resources
  - coordinated input and output
  - lack of progress (deadlocks and livelocks)
  - synchronization: access to the same data item by several processes/threads (typical example, deposits into account)

Synchronization Example

- At time t=6 T2 requests the withdrawal of $200 (and gets preempted); at t=8 T1 requests the deposit of $200; t=9, T1 reads balance ($300); at t=10, T2 adds $200, at t=11, T1 writes new balance; at t=12, T2 resumes and reads balance; at t=13, T2 subtracts $200; at t=14, T2 writes new balance



- What if order was changed: 6, 8, 9, 12, 13, 14, 10, 11?
- Other combinations?

More on Synchronization

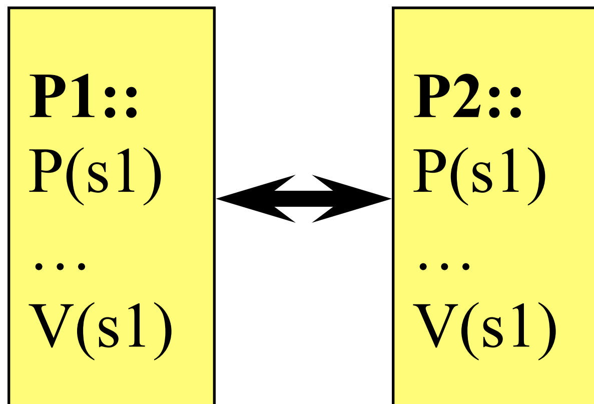
- Semaphores and primitives serve to achieve mutual exclusion and synchronized access to resources
- Clearly, there is more delays associated with attempting to access a resource protected by semaphores
- Non-preemptive scheduling solves that problem
- As an aside:
  - Which scheduling ***mechanism*** is more efficient? Preemptive or non-preemptive?
  - Within each type of scheduling (pr or non-pr), one can choose which ***policy*** he/she wants to use

Using Semaphores

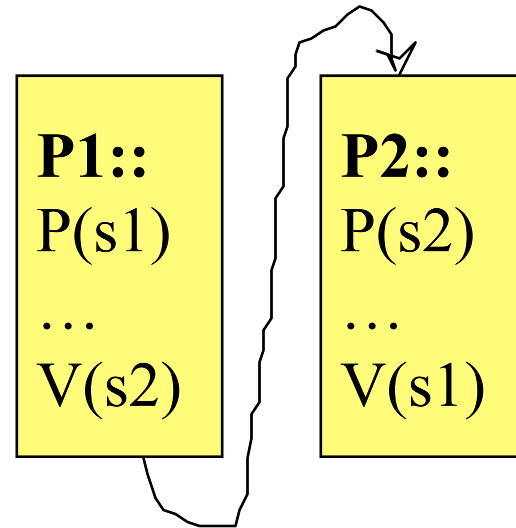
- When threads use semaphores, they are attempting to reserve a resource for usage. Only the threads that succeed are allowed in the CS.
- If there is a thread in the CS already, other requesting threads are blocked, waiting on an *event*
- When the thread exits the CS, the OS *unblocks* the waiting thread, typically during the V(s) sys_call
- The now unblocked thread becomes *ready*
- The OS may decide to invoke the scheduler… or not

Types of Synchronization

- There are 2 basic types of sync:



MUTEX



BARRIER SYNC

Deadlocks

- When a program is written carelessly, it may cause a…
- DEADLOCK!!!
- Each process will wait for the other process indefinitely
- How can we deal with these abnormalities?
- *Avoidance, prevention, or detection and resolution*
- Which one is more efficient?

```
P1::
P(s1)
P(s2)
…
V(s2)
V(s1)
```

```
P2::
P(s2)
P(s1)
…
V(s1)
V(s2)
```

Dining Philosophers

- There were some hungry philosophers, and some angry philosophers (not the same, not Rastas)
- Each needed two forks, each shared both his/her forks
- Possible deadlock situation! HOW?
- Is it possible to have a (literally) starvation situation?
- What is the best solution?
  - Round-robin? FIFO? How good are FIFO and RR?
- **Important**: what is the metric to judge a solution by?
  - Least overhead? Minimum starvation?
  - How to evaluate fairness?