Processes in Unix, Linux, and Windows

Professor Hugh C. Lauer
CS-3013, Operating Systems

(Slides include copyright materials from *Operating Systems: Three Easy Step*, by Remzi and Andrea Arpaci-Dusseau, from *Modern Operating Systems*, by Andrew S. Tanenbaum, 3rd edition, and from other sources)

Processes in Unix, Linux, and Windows

- In previous topic, we used "*process*" in a generic way to represent the abstraction of concurrency
- Unix pre-empted generic term "*process*" to mean something very specific
- Linux, Windows, and other OSes adopted Unix definition

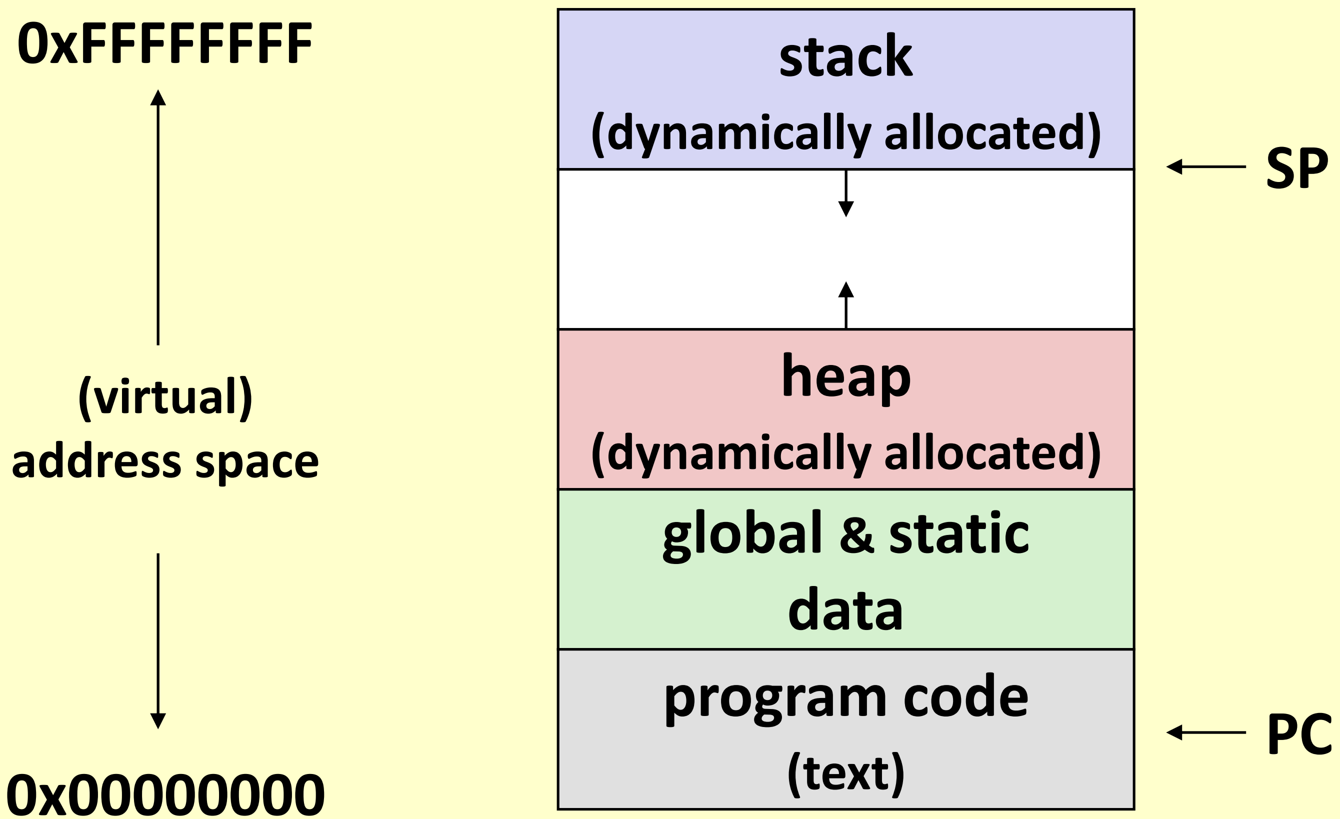
Process in Unix-Linux-Windows comprises

- **an *address space* – usually protected and virtual – mapped into memory**
- **the *code* for the running program**
- **the *data* for the running program**
- **an *execution stack* and *stack pointer* (SP); also *heap***
- **the *program counter* (PC)**
- **a set of processor *registers* – general purpose and status**
- **a set of system *resources***
  - files, network connections, pipes, …
  - privileges, (human) user association, …
- **…**

Reading Assignment

- **OSTEP**
  - Chapters 4, 5, and 6
- **(optional) R. Love, *Linux Kernel Development***
  - Chapter 3, thru p. 33

Process Address Space (traditional Unix)



Processes in the OS – Representation

- **To users (and other processes) a process is identified by its *Process ID* (PID)**
- **In the OS, processes are represented by entries in a *Process Table* (PT)**
  - PID is index to (or pointer to) a PT entry
  - PT entry = *Process Control Block* (PCB)
- **PCB is a large data structure that contains (or points to) all info about the process**
  - Linux – defined in `task_struct` (over 70 fields)
    - Some of which point to other data structures
    - See `include/linux/sched.h`
  - Windows XP – defined in *EPROCESS* – about 60 fields

Processes in the OS – PCB

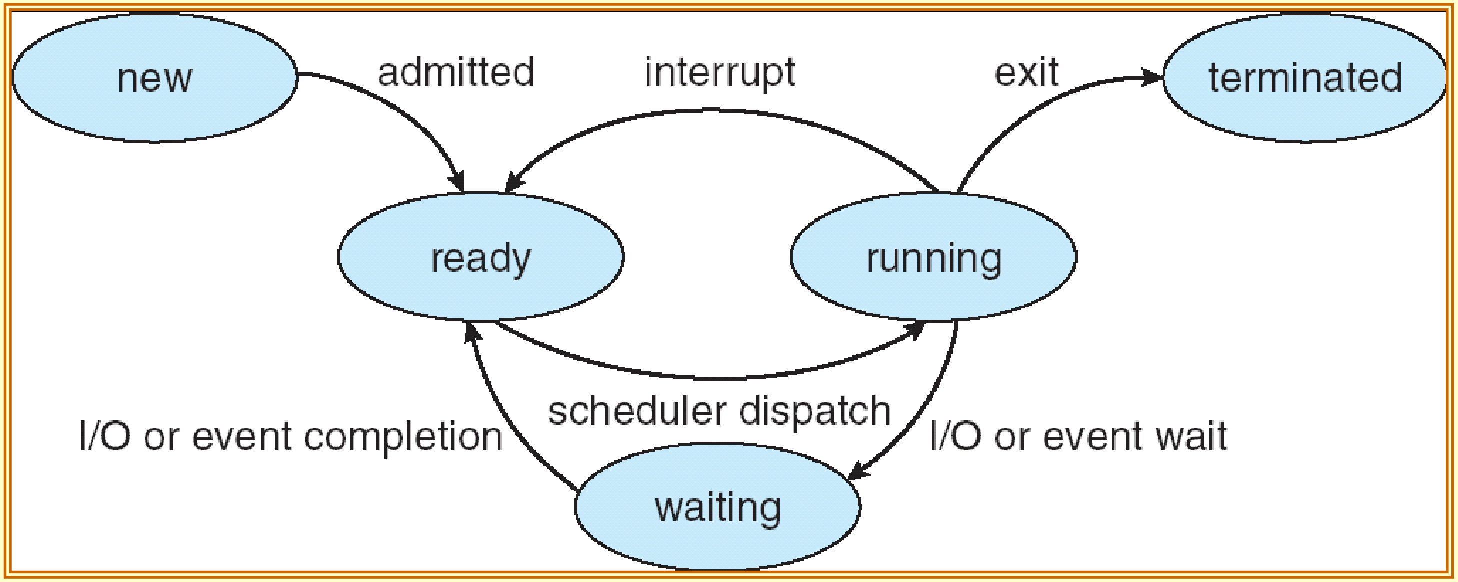
- **Typical PCB contains:**
  - execution state
  - PC, SP & processor registers – stored when process is not in *running* state
  - memory management info
  - privileges and owner info
  - scheduling priority
  - resource info
  - accounting info

Process – starting and ending

- **Processes are created …**
  - When the system boots
  - By the actions of another process (more later)
  - By the actions of a user (via another process)
- **Processes terminate …**
  - Normally – exit
  - Voluntarily on an error
  - Involuntarily on an error
  - Terminated (killed) by action of
    - a user or
    - another process

Processes – States

- **Process has an execution state**
  - *ready*: waiting to be assigned to CPU
  - *running*: executing on the CPU
  - *waiting*: waiting for an event, e.g. I/O



Processes – State Queues

- **The OS maintains a collection of *process state* queues**
  - typically one queue for each state – e.g., ready, waiting, …
  - each PCB is put onto a queue according to its current state
  - as a process changes state, its PCB is unlinked from one queue, and linked to another
- **Process state and the queues change in response to events – interrupts, traps**

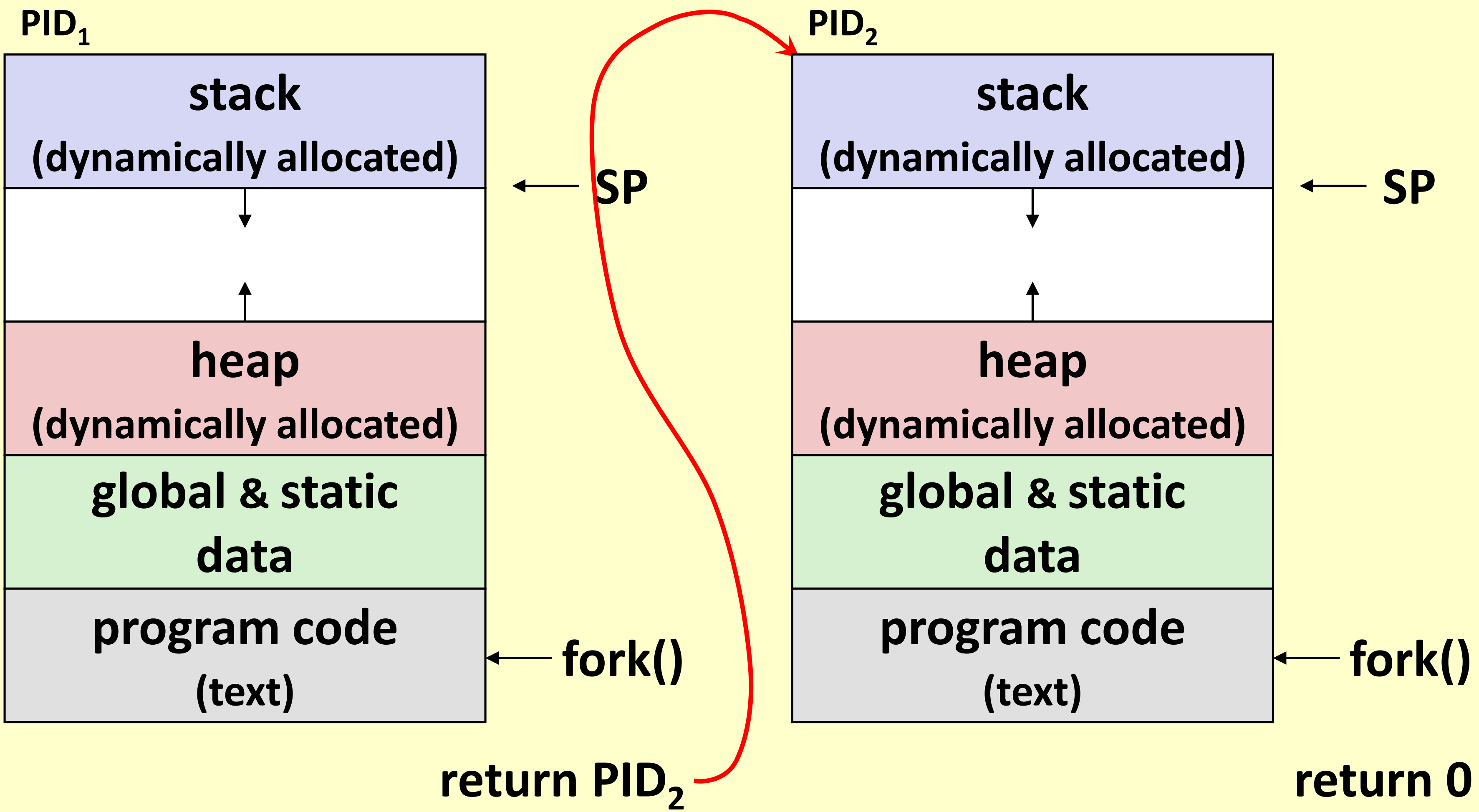
Processes – Privileges

- **Users are given privileges by the system administrator**
- **Privileges determine what *rights* a user has for an *object*.**
  - Unix/Linux – Read|Write|eXecute by user, group and "other" (i.e., "world")
  - Windows NT/XP/7 – *A*ccess *C*ontrol *L*ist
- **Processes "inherit" privileges from user**
  - or from creating process

Process Creation – Unix & Linux

- **Create a new (child) process – `fork();`**
  - Allocates new PCB
  - Clones the calling process (almost exactly)
    - Copy of calling process address space
    - Copy of all registers, condition codes, etc.
    - Copies resources in kernel (e.g. pointers to files)
  - Places new PCB on *Ready queue*
  - Return from **`fork()`** call
    - 0 for child
    - child PID for parent

Process Creation – Unix & Linux



Example of *fork( )*

```
int main(int argc, char **argv)
{
  char *name = argv[0];

  int child_pid = fork();

  if (child_pid == 0) {
    printf("Child of %s sees PID of %d\n",
           name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. My child is %d\n",
           name, child_pid);
    return 0;
  }
}
```

```
% ./forktest
Child of forktest sees PID of 0
I am the parent forktest. My child is 486
```

Result – Two identical processes

Parent

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s sees PID"
       " %d\n", name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. "
       "My child is %d\n", name,
          child_pid);
    return 0;
  }
}
```

False

Child

```
int main(int argc, char **argv)
{
  char *name = argv[0];
  int child_pid = fork();
  if (child_pid == 0) {
    printf("Child of %s sees PID"
           " %d\n", name, child_pid);
    return 0;

  } else {
    printf("I am the parent %s. "
           "My child is %d\n", name,
                  child_pid);
    return 0;
  }
}
```

True

Only difference

Questions?

Starting New Programs

- **Unix & Linux:–**
  - **`int exec (char *prog, char **argv)`**
  - Check privileges and file type
  - Loads program at path *prog* into address space
    - Replacing previous contents of address space!
    - Execution starts as function call to **`main()`**
  - Initializes context – e.g. passes arguments
    - **`*argv`**
  - Place PCB on *ready queue*
  - Preserves, pipes, open files, privileges, etc.

Executing a New Program (Linux-Unix)

- `fork()` followed by `exec()`

- Creates a new process as clone of previous one
  - I.e., same program, but different execution of it

- First thing that clone does is to replace itself with new program

Fork + *Exec* – shell-like

```
int main(int argc, char **argv)
{ char *argvNew[5];
  int pid;
  if ((pid = fork()) < 0) {
              printf( "Fork error\n");
              exit(1);
  } else if (pid == 0) { /* child process */
              argvNew[0] = "/bin/ls";  /* i.e., the new program */
              argvNew[1] = "-l";
              argvNew[2] = NULL;
              if (execvp(argvNew[0], argvNew) < 0) {
                 printf( "Execvp error\n");
                 exit(1);  /* program should not reach this point */
              }
       } else { /* parent */
              wait(pid); /* wait for the child to finish */
       }
}
```

*Wait*ing for a Process

- **Multiple variations of *wait* function**
  - Including non-blocking *wait* functions
- **Waits until child process terminates**
  - Acquires termination code from child
  - Child process is destroyed by kernel
- ***Zombie:*– a process that had never been *wait*ed for**
  - Hence, cannot go away!
  - See Love, *Linux Kernel Development*, pp 37-38
  - See OSTEP, §4.5, p. 32

Processes – Windows

- **Windows NT/XP/10 – combine `fork` & `exec`**
  - `CreateProcess(10 arguments)`
  - No parent child relationship
  - *Note* – privileges are required to create a new process

  - Much more about processes, threads, etc., in Windows than we can cover in this course

Traditional Unix

- ***Processes*** **are in *separate* address spaces**
  - By default, no shared memory
- ***Processes*** **are unit of scheduling**
  - A process is *ready*, *waiting*, or *running*
- ***Processes*** **are unit of resource allocation**
  - Files, I/O, memory, privileges, …
- ***Processes*** **are used for (almost) everything!**

Windows and Linux

- *Threads* (next topic) are units of scheduling
- Threads are used for everything

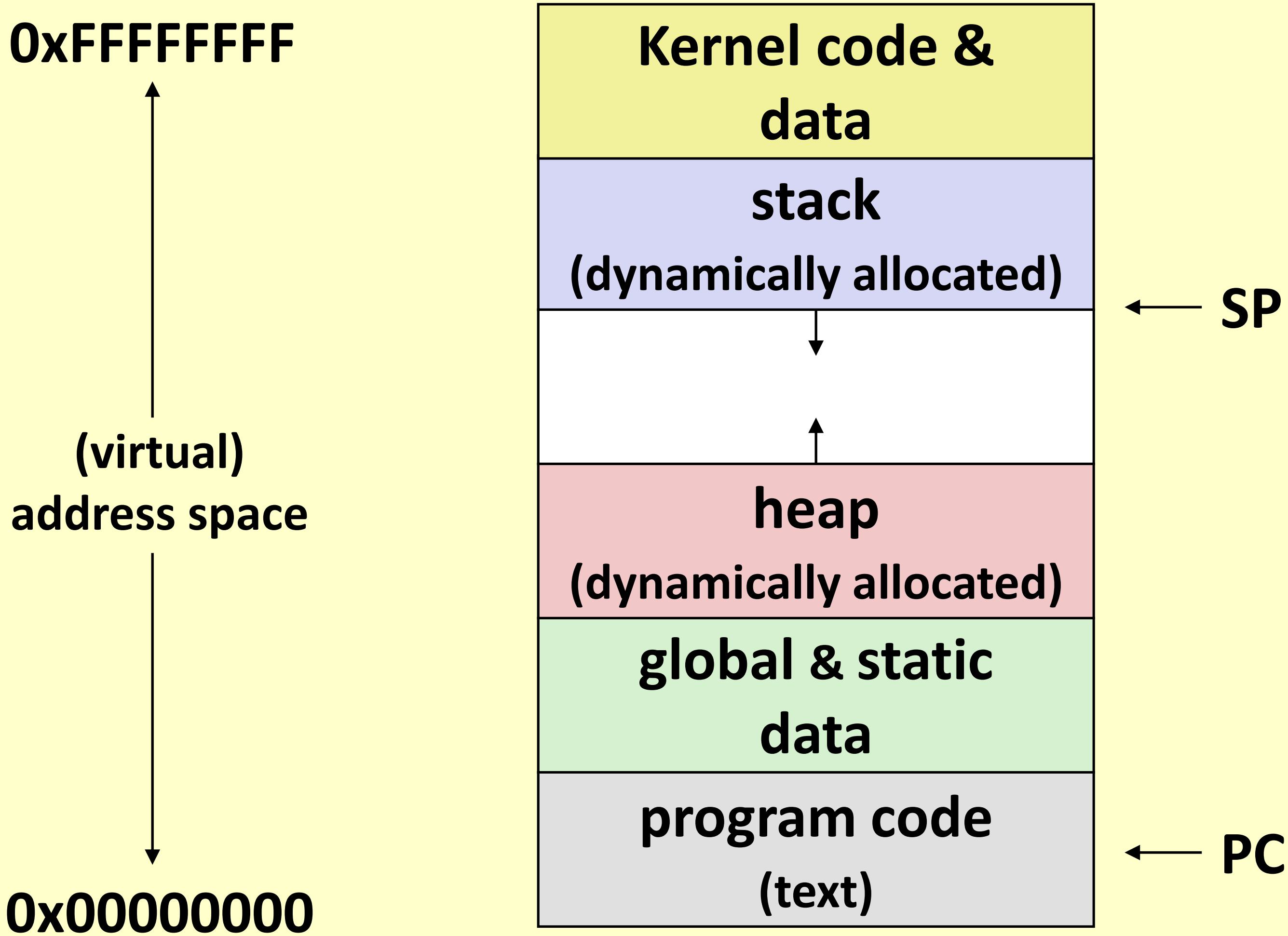
Non-Traditional Unix — e.g., iPhone, Android, etc.

- *Processes* are in *separate* address spaces
  - By default, no shared memory
- ***Processes* are unit of scheduling**
  - A process is *ready*, *waiting*, or *running*
- *Processes* are unit of resource allocation
  - Files, I/O, memory, privileges, …
- *Processes* are used for (almost) everything!

A Note on Implementation

- **Many OS implementations include (parts of) *kernel* in every address space**
  - Protected
  - Easy to access
  - Allows *kernel* to see into client processes
    - Transferring data
    - Examining state
    - …

Process Address Space (with kernel)



Linux Kernel Implementation

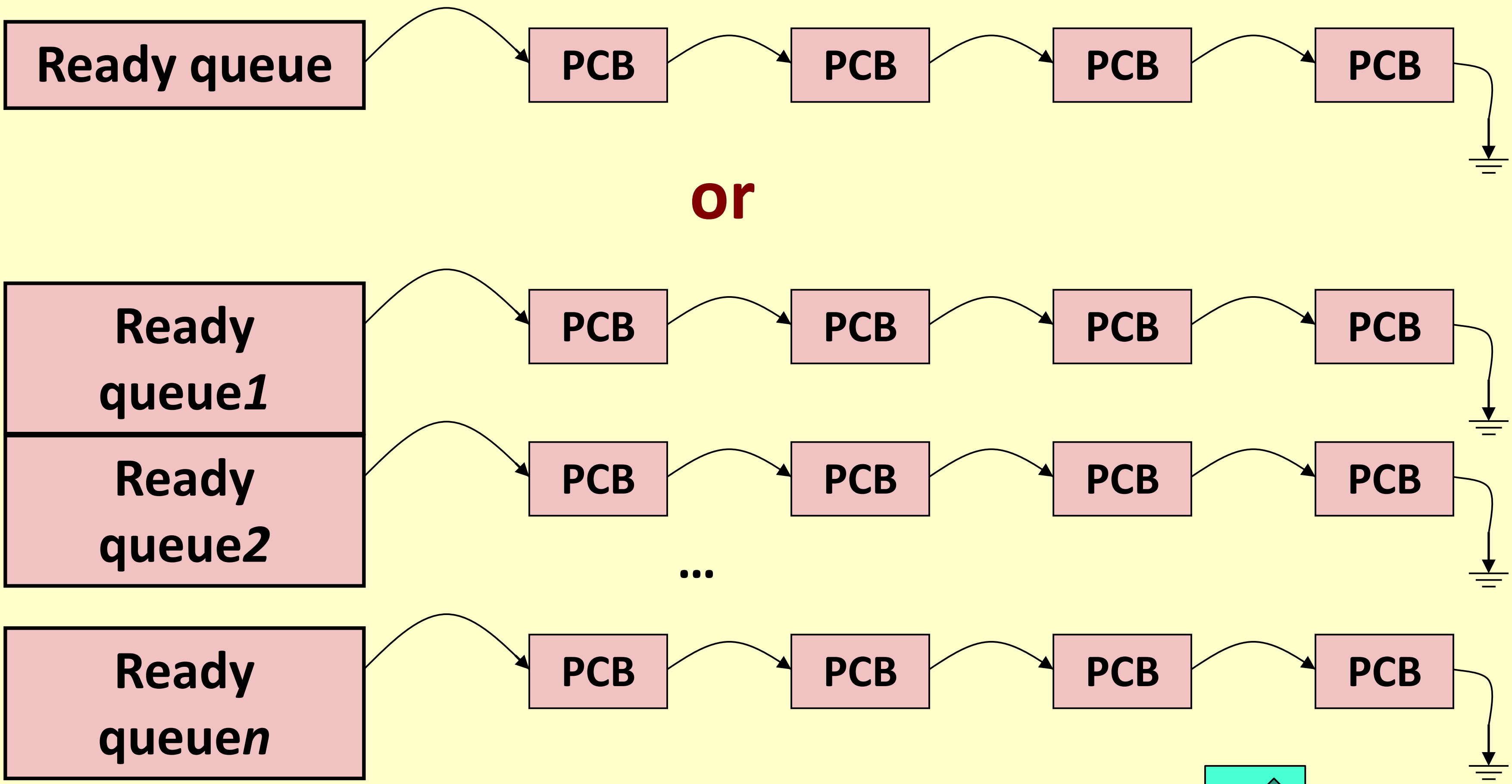
- **Kernel may execute in either *Process context* vs. *Interrupt context***
- **In *Process context*, kernel has access to**
  - Virtual memory, files, other process resources
  - May sleep, take page faults, etc., on behalf of process
- **In *Interrupt context*, no assumption about what process was executing (if any)**
  - No access to virtual memory, files, resources
  - May not sleep, take page faults, etc.

Processes in Other Operating Systems

- Implementations will differ
- Sometimes a subset of *Unix/Linux/Windows*
  Sometimes quite different
- May have more restricted set of resources

- Often, specialize in real-time constraints

Implementation of Processes



Reading Assignment

- OSTEP, Chapters 4, 5, and 6

Questions?